NPS ARCHIVE
2000.03
LIM, C.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SIMULATION OF
SIGNALING SYSTEM NO. 7
MESSAGE TRANSFER PART 2

by

Lim Chin Thong
March 2000

Thesis Advisor: John McEachen
Second Reader: Murali Tummala

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE
Simulation of Signaling System No. 7
Message Transfer Part 2

5. FUNDING NUMBERS

6. AUTHOR(S)
Lim, Chin Thong

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8.PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution unlimited.	A

13. ABSTRACT ***(maximum 200 words)***

The objective of this work is to perform simulation modeling of the Signaling System No.7 (SS7) network with particular emphasis on modeling of the Message Transfer Part Level 2. The basics of common channel signaling using Signaling System No. 7 is initially outlined and discussed with reference to the ITU-T Q.7xx-Series Recommendations. This includes the protocol stack, signaling points, signaling links and typical network structure. In particular, the functionality of the Message Transfer Part, which provides the main mechanism to convey signaling messages, is discussed in detail.

Subsequently the modeling of the Message Transfer Part, in particular MTP level 2, using the simulation tool OPNET from MIL3. Inc. is presented. The model uses a multi-layer modular approach, with each layer corresponding to the SS7 layer it is modeling. The functional blocks within each layer are thought of as processes. With their buffers and processors, these processes form a complex interlinked queuing model that is complicated to analyze but is readily simulated.

In order to illustrate the use of the simulation model, the basic linkset delay between two signaling points under a heavy traffic load is simulated and compared with analysis based on M/G/1 queuing models.

14. SUBJECT TERMS	13. NUMBER OF PAGES
Signaling, Signaling System No.7, Link Delay	*159
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICA-TION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited

SIMULATION OF SIGNALING SYSTEM NO. 7 MESSAGE TRANSFER PART 2

Lim, Chin Thong
B. Eng. National University of Singapore, 1994

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

ABSTRACT

The objective of this work is to perform simulation modeling of the Signaling System No.7 (SS7) network with particular emphasis on modeling of the Message Transfer Part Level 2. The basics of common channel signaling using Signaling System No. 7 is initially outlined and discussed with reference to the ITU-T Q.7xx-Series Recommendations. This includes the protocol stack, signaling points, signaling links and typical network structure. In particular, the functionality of the Message Transfer Part, which provides the main mechanism to convey signaling messages, is discussed in detail.

Subsequently the modeling of the Message Transfer Part, in particular MTP level 2, using the simulation tool OPNET from MIL3. Inc. is presented. The model uses a multi-layer modular approach, with each layer corresponding to the SS7 layer it is modeling. The functional blocks within each layer are thought of as processes. With their buffers and processors, these processes form a complex interlinked queuing model that is complicated to analyze but is readily simulated.

In order to illustrate the use of the simulation model, the basic linkset delay between two signaling points under a heavy traffic load is simulated and compared with analysis based on M/G/1 queuing models.

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
1. Common Channel Signaling 1
2. Performance Analysis 2
B. DISSERTATION OVERVIEW 2

II. SIGNALING SYSTEM 7 5
A. INTRODUCTION 5
B. SIGNALING POINTS AND SIGNALING LINKS 6
C. SS7 PROTOCOL STACK 8
D. MTP LEVEL 2 10
1. Types Of Signaling Units 10
2. The MTP Level 2 Functions 13
3. Error Detection with Flags, Zero Insertion and CRC Checks 14
4. Error Correction with Go Back-N Retransmission Algorithm 14
5. Transmission (TXC) and Reception Control (RC) 16
6. Error Rate Monitoring 17
7. Overall Link State Control and Initial Alignment 19
E. MTP LEVEL 3 20
1. The MTP Level 3 Functions 20
2. Routing Labels 22
3. Routing of Messages 23

III. OPNET MODELLING 25
A. GENERAL CAPABILITIES OF OPNET 25
B. MODELLING SIGNALING POINTS AND NETWORKS USING OPNET 26
C. MODELING MTP PROCESSES USING OPNET 29
D. MODELING MTP LEVEL 2 PROCESS 29
1. Model Overview 29
2. Buffers and Processor Delay 30
E. MODELLING OF MESSAGE TRANSFER PART LEVEL 3 32

IV. LINKSET DELAY USING QUEUE MODELING 33

A. INTRODUCTION 33
B. LOAD DISTRIBUTION IN A LINKSET 33
C. LINK SET DELAY MODELLING 34
1. The M/G/1 Queuing Model 34
2. General Distribution for Service Time 35
3. Average Link Delay Estimation 36
4. Average Linkset Delay Estimation 36

V. SIMULATION RESULTS AND COMPARSION 39
A. SCOPE OF SIMULATION 39
B. SIMULATION NETWORK CONFIGURATION 39
C. LINKSET DELAY COMPARISON 40

VI. CONCLUDING REMARKS 47

VII. APPENDIX: OPNET SOURCE CODE FOR MTP LEVEL 2 49
A. HEADER 49
B. STATE VARIABLES 57
C. TEMPORARY VARIABLES 62
D. FUNCTION DECLARATION 63
E. EXECUTIVE FOR "init" STATE 87
F. EXECUTIVE FOR "ext_msg" STATE 89
G. EXECUTIVE FOR "TXC_rcv" STATE 91
H. EXECUTIVE FOR "TX_MSU" STATE 92
I. EXECUTIVE FOR "TXC" STATE 95
J. EXECUTIVE FOR "RC_rcv" STATE 101
K. EXECUTIVE FOR "RC" STATE 106
L. EXECUTIVE FOR "LSC" STATE 109
M. EXECUTIVE FOR "IAC" STATE 122
N. EXECUTIVE FOR "POC" STATE 127
O. EXECUTIVE FOR "AERM" STATE 129
P. EXECUTIVE FOR "SUERM" STATE 131
Q. EXECUTIVE FOR "CC" STATE 133

LIST OF REFERENCES 134

INITIAL DISTRIBUTION LIST 137

LIST OF FIGURES

Fig 2.1 SS7 Signaling Points 6

Fig 2.2 A Typical SS7 Network 8

Fig 2.3 SS7 Protocol Stack and The OSI Reference Model 9

Fig 2.4 SS7 Signal Units 11

Fig 2.5 Functional Block Diagram of MTP Level 2 14

Fig 2.6 Use of Sequence Numbers and Indicator Bits in SS7 15

Fig 2.7 Functional Procedure of AERM and SUERM 18

Fig 2.8 Functional Block Diagram of MTP Level 3 21

Fig 2.9 ANSI vs. ITU-T SIO and SIF Format 22

Fig 3.1 Main Types of OPNET Modules 25

Fig 3.2 Processor-Buffer Model of an SSP with 2 Links 27

Fig 3.3 A 2 Link SSP with ISDN User Part 27

Fig 3.4 A 4 Link STP 28

Fig 3.5 A Typical 2 SSP SS7 Network 28

Fig 3.6 Process Model of MTP Level 2 31

Fig 4.1 Single Stage Queuing Model for Link Set 34

Fig 4.2 Queuing Theory Prediction of Link Set Delay (for 4 bit SLS) 37

Fig 5.1 2 SSP Network Used for Simulation 39

Fig 5.2 Average Link Delay Comparison Between Simulation and Estimation 41

Fig 5.3 Average Linset Delay at Various MTP-3 Service Interval 43

Fig 5.4 MTP-2 Link Congestion at Traffic Load of 1.96 44

Fig 5.5 MTP-3 Routing Function Delay with MTP-2 congested at Traffic Load of 1.96 44

Fig 5.6 MTP-3 Routing Function Congestion at Traffic Load of 1.63 45

Fig 5.7 MTP-2 Link delay with MTP-3 congested at Traffic Load of 1.63.....45

LIST OF TABLES

Table 2.1 Basic MTP Level 2 Fields 11

Table 2.2 Link Status 12

Table 2.3 Service Indicator Values 12

Table 2.4 Signaling Unit Transmission Order 17

Table 2.5 MTP Level 2 Timers 17

Table 2.6 Recommended Parameters for AERM and SUERM 19

Table 2.7 MTP Level 2 Link Startup Procedure 20

Table 3.1 Major OPNET Version 6.0 Components 26

Table 3.2 MTP Level 2 buffers 31

Table 4.1 n_H, n_L, λ_H and λ_L for 4 bit SLS 34

Table 5.1 Simulation Parameters 40

LIST OF SYMBOLS

K	No. of links in a linkset
N_s	No. of possible SLS codes
n_H	No. of links with higher traffic load
n_L	No. of links with lower traffic load
λ_{LS}	Total traffic load in linkset
λ_H	Higher link traffic load
λ_L	Lower link traffic load
x	Message Length
$\bar{x}$	Average Message Length
p_k	Probability of message having length x
C_x	Coefficient of variation for Message Length Distribution
C	Link Data Rate
ρ	Link utilization
T_t	Message transmission time (time required to transmit message at data rate)
T_s	Message service time (or message waiting time in the buffer)
T_p	Message propagation time
T_{LS}	Average linkset delay
T_d	Average link delay
L	Distance between two signaling points
v	Speed of transmission medium
$\overline{N}$	Average No. of MSUs transmitted

LIST OF ABBREVIATIONS AND ACRONYMS

AERM	Alignment Error Rate Monitor
AIN	Advanced Intelligent Network
ANSI	American National Standards Institute
BIB	Backward Indicator Bit
B-ISDN	Broadband Integrated Services Digital Network
BSN	Backward Sequence Number
CC	Congestion Control
CK	Checksum
DAEDR	Delimitation, Alignment and Error Detection (Receiving)
DAEDT	Delimitation, Alignment and Error Detection (Transmission)
DPC	Destination Point Code
DUP	Data User Part
ETSI	European Telecommunications Standards Institute
F	Flag
FIB	Forward Indicator Bit
FISU	Fill-In Signaling Unit
FSN	Forward Sequence Number
GTT	Global Title Translation
IAC	Initial Alignment Control
ISDN	Integrated Services Digital Network
ISUP	ISDN User Part
ISO	International Standards Orgainization
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
LI	Length Indicator
LSC	Link State Control
LSSU	Link Status Signaling Unit
MTP	Message Transfer Part
MTP-2	Message Transfer Part Level 2
MTP-3	Message Transfer Part Level 3
MSU	Message Signaling Unit

OMAP	Operation, Maintenance and Administration Part
OPC	Originating Point Code
OSI	Open Systems Interconnect
PC	Point Code
PCS	Personal Communication Systems
POC	Processor Outage Control
PSTN	Public Switched Telephone Networks
RC	Receiver Control
SCCP	Signaling Connection Control Part
SCP	Signaling Control Point
SF	Status Field
SI	Service Indicator
SIF	Signaling Information Field
SIO	Service Information Octet
SLS	Signaling Link Selection
SMH	Signaling Message Handling
SNM	Signaling Network Management
SP	Signaling Point
SPC	Stored Program Control
SS7	Signaling System No. 7
SSF	Sub-Service Field
SSP	Service Switching Point
STP	Signaling Transfer Point
SU	Signaling Unit
SUERM	Signaling Unit Error Rate Monitor
TCAP	Transaction Capabilities Application Part
TUP	Telephone User Part
TXC	Transmission Control

ACKNOWLEDGEMENT

The author wants to thank Prof. McEachen for his guidance, advice and patience during the work of this thesis. His continuous help and undivided support in getting the various programming tools, OPNET license, laptops and technical books is much appreciated. He is always there when his presence is needed.

The author would also want to thank Prof. Tummala for contributing his time and effort in reviewing this thesis. His well conducted lessons in high speed networking has proved to be very valuable in the development of this work.

Lastly, I would devote my work to my wife. Life would never be the same without her support and understanding throughout my stay in Naval Postgraduate School.

I. INTRODUCTION

The objective of this work is to perform a multi-layered discrete event modeling of the SS7 network for simulation. Detailed representation of functional procedures for the Message Transfer Part will be incorporated to the maximum extent possible. To further illustrate the use of the model, Link Delay simulations using the model are performed and compared with results obtained from a conventional queuing model.

A. BACKGROUND

1. Common Channel Signaling

Since the invention of the telephone in 1876, signaling has been linked to analog circuit switching in the Public Switched Telephone Networks (PSTN). Signaling messages were initially multiplexed on a single line, using separate channels from voice called In-Band signaling. However, later introduction of Stored Program Control (SPC) exchanges followed by digital exchanges allowed the possibility of applying digital computer communication techniques to provide a dedicated data communication network for the transfer of signaling information between exchanges. This form of signaling is called "Common Channel Signaling".

In 1976, CCITT began work on standardizing a common channel signaling standard called Signaling System No.7 (SS7). This standard underwent several stages of refinement. In succession appeared, the Yellow Book Recommendations in 1981, the Red Book in 1985 and the Blue Book in 1989 prior to the present ITU-T Q.7xx-Series Recommendations.

Today, SS7 has been widely implemented, forming the control structure behind the modern day telecommunications network. Due to SS7's reliability and ability to support non-circuit related services, its application has not been limited to just call connections in telephone networks but has since been extended to providing signaling services in Integrated Services Digital Network (ISDN), Broadband ISDN, Advance Intelligent Network (AIN) and also Mobile Telephone systems.

2. Performance Analysis

Conventional performance analysis of the constantly varying modern day telecommunication networks using statistical models and queuing theory has proven to be complicated and restrictive, especially when networks become increasingly large and changes are often. Discrete event modeling and simulation offers a new approach to network analysis, especially when quick impact studies are required for the introduction of new services, new routing algorithms or new equipment.

William and Kuhn [1] modeled the Blue Book specifications of SS7 by representing the major functional elements in the specifications with servers and queues to form a multi-layered model. Unfortunately, the model is limited to a steady state network analysis as stationary conditions using Markovian queuing theory are assumed. In Unger, Goetz and Maryka [2], the dynamic analysis of large SS7 systems using discrete event modeling is addressed. The improvement presented in this thesis is achieved through detailed incorporation of the message handling and network management procedures of SS7.

B. DISSERTATION OVERVIEW

In this work, the details of SS7 are first discussed in Chapter II. The notion of signaling points, signaling links and a signaling network are explained followed by a detail description of the Message Transfer Part. The basic procedures of the various functional blocks in the layer are explained. This includes alignment, error correction and detection, and routing.

The use of the simulation tool, OPNET, to model the Message Transfer Part is discussed in Chapter III. The implementation of the various functional procedures, buffers and processors (servers) in OPNET are explained. As an illustration, a two-signaling-point network is constructed for link delay analysis under heavy traffic load conditions. For comparison, the link delay analysis based on Markovian queueing models is also derived in Chapter IV. Chapter V then summarizes the simulation results and compares them with that obtained using queuing models.

Finally, Chapter VI provides the concluding remarks involved in the simulation modeling of SS7, and Appendix I contains the C source code for the OPNET model.

THIS PAGE IS INTENTIONALLY LEFT BLANK

II. SIGNALING SYSTEM 7

A. INTRODUCTION

SS7 Q.7xx-Series Recommendations is a global Common Channel standard for telecommunications defined by the International Telecommunication Union Telecommunication Standardization Sector (ITU-T). The recommendation mainly defines the procedures and protocol by which the Public Switched Telephone Network (PSTN) exchange signaling information over a separate dedicated digital network to effect wired and wireless (cellular) call setup, routing and control. From the ITU definition of SS7, there are also several national variants such as the American National Standards Institute (ANSI) and Bell Communications Research (Bellcore) standards used in North America and the European Telecommunications Standards Institute (ETSI) standard used in Europe. Nevertheless these variants differ slightly only in format, and their compatibilities are maintained at the international level. Henceforth, the ITU-T recommendations will be used as the main reference for further discussion in this thesis.

As mentioned, the use of SS7 was originally intended for PSTN networks; however, due to its high emphasis on reliability and availability[1], the use of SS7 has been extended to provide signaling services to ISDN and AIN systems. It appears that it will conitnue in this role even with the expansion of wireless networks and the introduction of Broadband ISDN networks. Some of the most commonly used services of SS7 are:

- Basic call setup, management, and tear down.
- Wireless services, such as personal communications services (PCS), wireless roaming, and mobile subscriber authentication.
- Toll-free (800/888) and toll (900) wired services.
- Enhanced call features, such as forwarding, caller ID and three-way calling.
- Efficient and secure worldwide telecommunications.

---

[1] SS7 has set a high availability objective of 99.9998% for any signaling route. This corresponds to a maximum permissible downtime of only 10 minutes per year.

B. SIGNALING POINTS AND SIGNALING LINKS

Signaling in SS7 basically involves the transmission of data packets called Signaling Units (SU) via a digital network comprised of nodes called Signaling Points (SP). All signaling traffic originates and terminates at a signaling point. As shown in Fig 2.1, there are three types of signaling points, namely Service Switching Point (SSP), Signaling Transfer Point (STP) and Service Control Point (SCP). Each signaling point in the SS7 network is uniquely identified by a numeric point code. Point codes are carried in signaling messages to identify the source and destination of each message. Each signaling point uses a routing table to select the appropriate signaling path for each message.

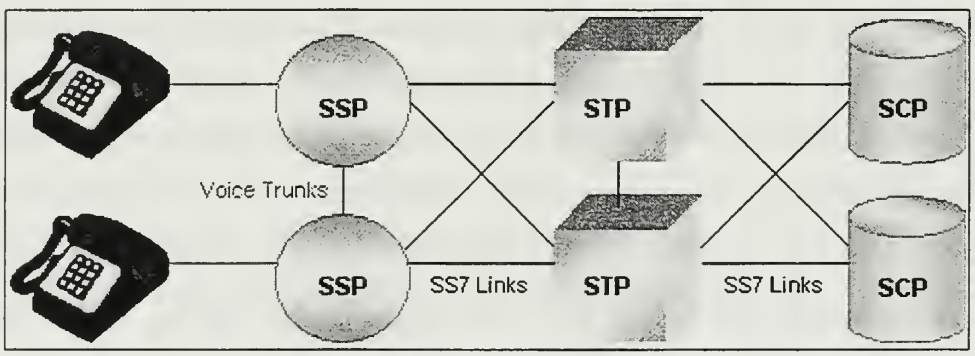

Fig 2.1 SS7 Signaling Points

SSPs are switches that originate, terminate, or tandem calls. An SSP sends signaling messages to other SSPs to setup, manage, and release voice circuits required to complete a call. An SSP may also send a query message to a centralized database (an SCP) to determine how to route a call (e.g., a toll-free 1-800/888 call in North America). An SCP sends a response to the originating SSP containing the routing number(s) associated with the dialed number. An alternate routing number may be used by the SSP if the primary number is busy or the call is unanswered within a specified time. Actual call features vary from network to network and from service to service.

In more complicated networks, traffic between signaling points may be routed via a packet switch called an STP. An STP routes each incoming message to an outgoing signaling link based on routing information contained in the SS7 message. Because it acts as a network hub, an STP provides improved utilization of the SS7 network by eliminating the need for direct links between signaling points.

The signaling links between signaling points are formed by bi-directional channels. The most common data rate used on a signaling channel is 56 or 64 kbps. Although higher data rates can be used, the 56/64 kbps bandwidth could still adequately serve the signaling traffic in modern day SS7 networks. Typically, more bandwidth is achieved by bundling multiple links. When more than one signaling link exists between two SPs, it is called a Linkset.

Unlike earlier signaling networks, SS7 signaling links are dedicated to carrying only signaling traffic. All signaling occurs out-of-band on these dedicated channels rather than sharing in-band with the voice channels. Hence it is also widely called common channel signaling. Compared to in-band signaling, out-of-band signaling provides:

- Faster call setup times,
- More efficient use of voice circuits,
- Support for Intelligent Network (IN) services which require signaling to network elements without voice trunks (e.g., database systems), and
- Improved control over fraudulent network usage.

Because the SS7 network is critical to call processing, SCPs and STPs are usually deployed in mated pair configurations in separate physical locations to ensure network-wide service in the event of an isolated failure. Links between signaling points are also provisioned in pairs. Traffic is shared across all links in the linkset. If one of the links fails, the signaling traffic is rerouted over another link in the linkset. The SS7 protocol provides both error correction and retransmission capabilities to allow continued service in the event of signaling point or link failures. Fig 2.2 illustrates the separation of signaling links from voice trunks between exchanges (symbol "Ex"), and the use of mated pairs for STPs and SCPs to achieve redundancy. The mesh structure or quad structure formed by the two mated pairs of STPs are commonly used in North America.

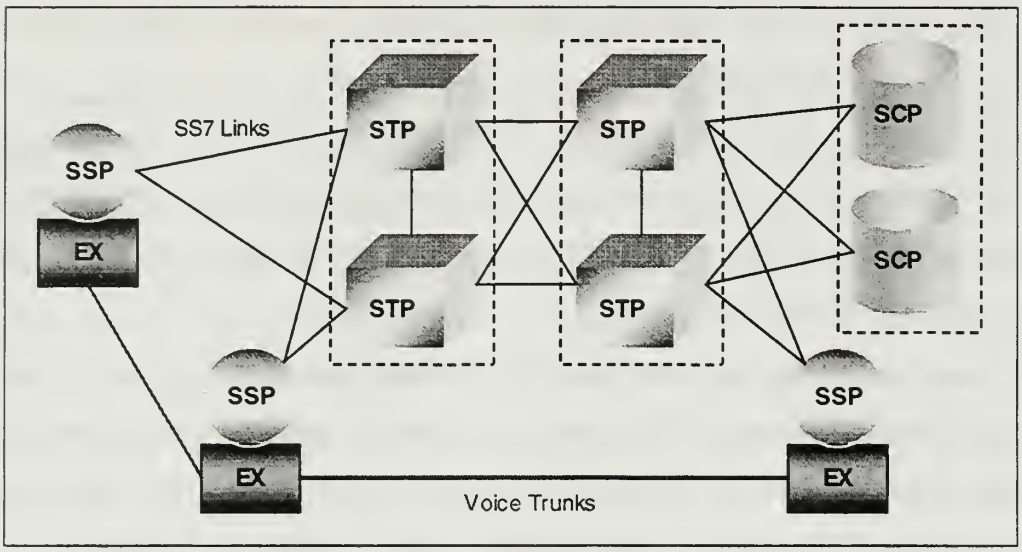

Fig 2.2 A Typical SS7 Network

C. SS7 PROTOCOL STACK

Like all network protocols, the hardware and software functions of the SS7 protocol are divided into functional levels [3]. The relation between these levels as compared to the Open Systems Interconnect (OSI) 7-layer model defined by the International Standards Organization (ISO) is illustrated in Fig 2.3.

Among the SS7 layers, the layers above the Message Transfer Parts (MTP) are considered the user layers, which generate signaling units for their own purposes. The role of the MTP layers is to ensure signaling units sent by the upper layers are delivered accurately and efficiently to their destination. The Message Transfer Part is divided into three levels with MTP Level 1, 2 and 3 being equivalent to the OSI Physical, Data Link and Network Layers, respectively. The lowest level, MTP Level 1, defines the physical, electrical, and functional characteristics of the digital signaling link. Physical interfaces defined include E-1 (2048 kb/s; 32 64 kb/s channels), DS-1 (1544 kb/s; 24 64 kp/s channels), V.35 (64 kb/s), DS-0 (64 kb/s), and DS-0A (56 kb/s). More detail discussion on these layers will be presented in the subsequent sections.

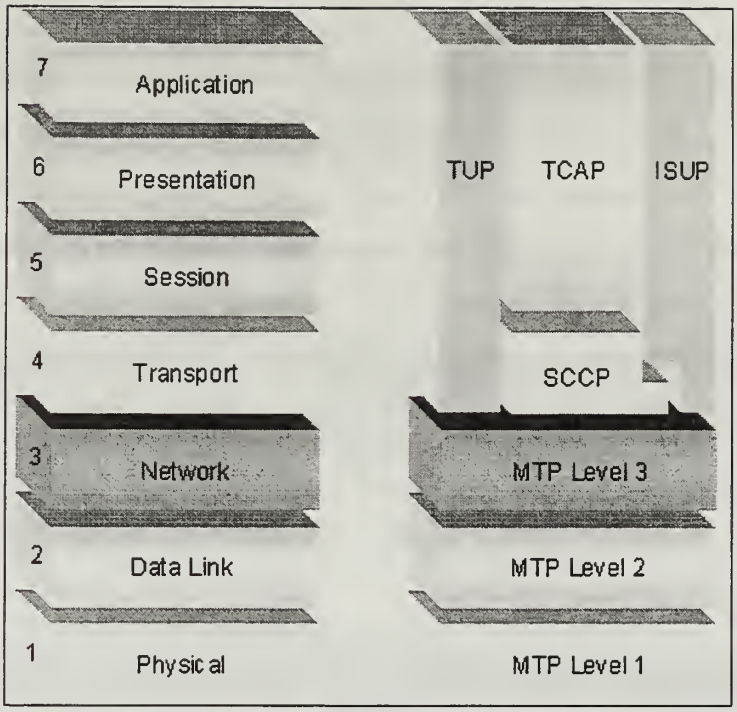

Fig 2.3 SS7 Protocol Stack and The OSI Reference Model

The ISDN User Part (ISUP) defines the protocol used to set-up, manage, and release trunk circuits that carry voice and data between terminating line exchanges (e.g., between a calling party and a called party). ISUP is used for both ISDN and non-ISDN calls. However, calls that originate and terminate at the same switch do not use ISUP signaling.

In some parts of the world (e.g., China and Brazil), the Telephone User Part (TUP) is used to support basic call setup and tear down. TUP handles analog circuits only. In most countries, ISUP has replaced TUP for call management.

Signaling Connection Control Part (SCCP) provides connectionless and connection-oriented network services and global title translation (GTT) capabilities above MTP Level 3. A global title is an address (e.g., a dialed 800 number, calling card number, or mobile subscriber identification number), which is translated by SCCP into a destination point code and subsystem number. A subsystem number uniquely identifies an application at the destination signaling point. SCCP is used as the transport layer for TCAP-based services.

Transaction Capabilities Application Part (TCAP) supports the exchange of non-circuit related data between applications across the SS7 network using the SCCP connectionless service. Queries and responses sent between SSPs and SCPs are carried in TCAP messages. For example, an SSP sends a TCAP query to determine the routing number associated with a dialed 800/888 number and to check the personal identification number (PIN) of a calling card user. In mobile networks (IS-41 and GSM), TCAP carries Mobile Application Part (MAP) messages sent between mobile switches and databases to support user authentication, equipment identification, and roaming.

Lastly, Operations, Maintenance and Administration Part (OMAP) is allocated for future definition. Presently, OMAP services may be used to verify network routing databases and to diagnose link problems.

D. MTP LEVEL 2

1. Types Of Signaling Units

The role of MTP Level 2 [4] is to ensure accurate transmission of a message across a signaling link between two adjacent signaling points. To achieve this, MTP-2 implements flow control, message sequence validation, and error checking. When an error occurs on a signaling link, the message (or set of messages) is retransmitted. There are three kinds of signal units: Fill-In Signal Units (FISUs), Link Status Signal Units (LSSUs), and Message Signal Units (MSUs) (Fig 2.4). Each type of signaling unit is used for a specific purpose; however, all of them have the basic level 2 fields as listed in able 2.1.

Fill-In Signal Units are the simplest signaling units in that they only carry the basic level 2 fields. FISUs are transmitted continuously on a signaling link whenever there are no other signal units (MSUs or LSSUs) to be sent. There are two main reasons for doing so. First, it provides acknowledgment to signaling units received when there are no LSSUs or MSUs to send. Second, since a FISU carries all the necessary level 2 fields, a correct receipt of a FISU indicates good link quality and vice versa. Hence FISUs provide a fast and responsive means of detecting link failures.

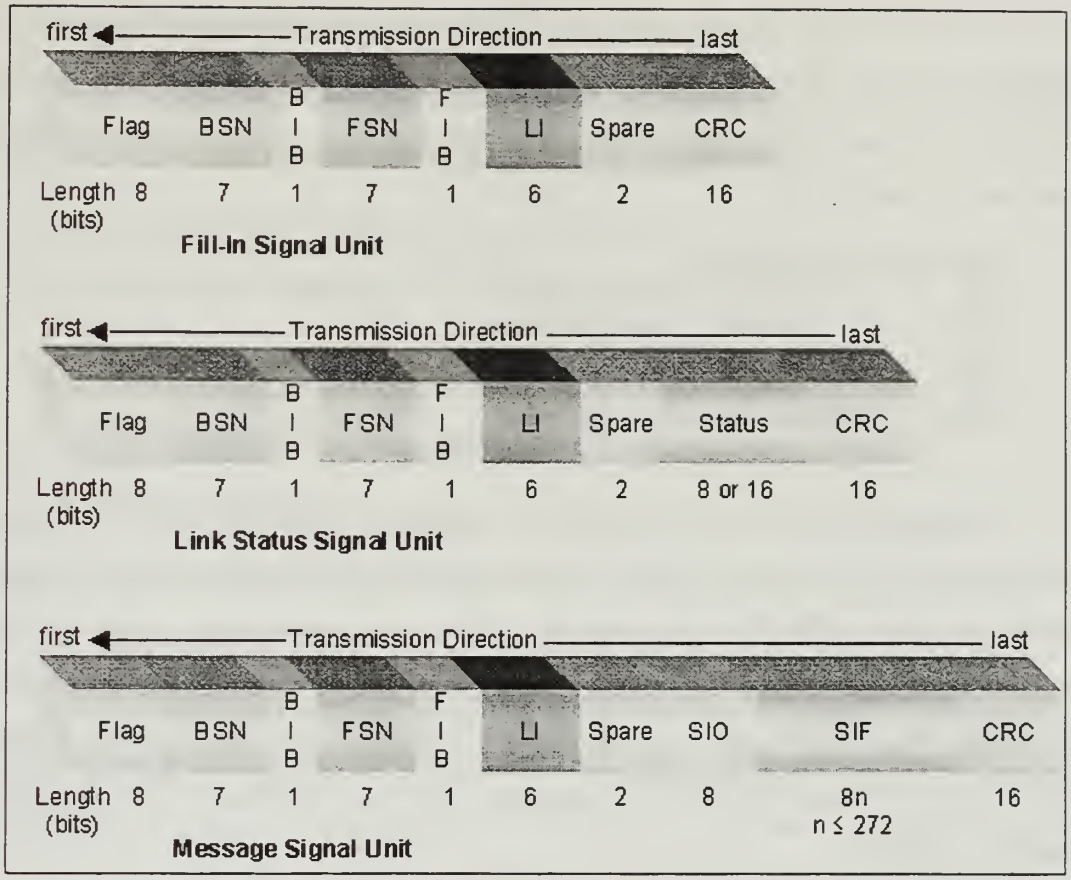

Fig 2.4 SS7 Signal Units

Field	Description
Flag	The flag indicates the beginning and end of signal unit. It is simply a 8 bit binary constant with the value **0111 1110**. Before transmitting a signal unit, MTP Level 2 removes "false flags" by adding a zero-bit after any sequence of five one-bits. Upon receiving a signal unit and stripping the flag, MTP Level 2 removes any zero-bit following a sequence of five one-bits to restore the original contents of the message.
Backward Sequence Number (BSN)	The BSN is used to acknowledge the receipt of signal units by the remote signaling point. The BSN contains the sequence number of the signal unit being acknowledged.
Backward Indicator Bit (BIB)	The BIB indicates a negative acknowledgement by the remote signaling point when toggled.
Forward Sequence Number (FSN)	The FSN contains the sequence number of the signal unit.
Forward Indicator Bit (FIB)	The FIB is used in error recovery like the BIB.
Length Indicator (LI)	This 6-bit field provides the length of the signaling unit after the LI field in octets. 0 for FISUs, 1 or 2 for LSSUs and 3 to 63 MSUs. The maximum length of a signal unit is 279 octets: 272 octets (data) + 1 octet (flag) + 1 octet (BSN + BIB) + 1 octet (FSN + FIB) + 1 octet (LI + 2 bits spare) + 1 octet (SIO) + 2 octets (CRC).

Table 2.1 Basic MTP Level 2 Fields

Link Status Signal Units carry one or two octets of link status information. The link status is used to control link alignment and to indicate the status of a signaling point (e.g., local processor outage) to the adjacent remote signaling point. The states of the link are defined in Table 2.2.

Link Status	Description
SIO	Link status: "Out of Alignment"
SIN	Link status: "Normal Alignment"
SIOS	Link status: "Out of Service"
SIE	Link status: "Emergency Alignment"
SIPO	Link status: "Processor Outage"

Table 2.2 Link Status

Message Signal Units are the main signaling information carriers. Signaling information of varying length from the upper user layers are carried in the Signaling Information Field (SIF). The type and structure of the information carried is usually differentiated by the Service Information Octet (SIO), which in turn is formed by a 4 bit Sub-Service Field (SSF) and a 4 bit Service Indicator (SI). The Service Indicator specifies the MTP user (Table 2.3), thereby allowing the decoding of the information contained in the SIF.

SI Value	MTP User
0	Signaling Network Management Message (SNM)
1	Signaling Network Tesing and Maintenance Messages
2	Spare
3	Signaling Connection Control Part (SCCP)
4	Telephone User Part (TUP)
5	ISDN User Part (ISUP)
6	Data User Part (call and circuit-related messages)
7	Data User Part (facility registration/cancellation messages)
8	Reserved for MTP Testing User Part
9	Broadband ISDN User Part
10	Satellite ISDN User Part
11-15	Spare

Table 2.3 Service Indicator Values

Besides, the SIF in an MSU usually starts with a routing label, which contains the source and destination address of the MSU. This label is used by STPs to determine the path through which the MSU is to be routed. LSSUs and FISUs contain neither a routing label nor an SIO as they are only sent between two adjacent signaling points. For more information about routing labels, refer to Section E for the description on MTP Level 3.

2. The MTP Level 2 Functions

ITU-T Q.703 describes the function and procedures relating to the transfer of signaling information over one signaling link. These functions include:

- Signal Unit Delimitation
- Signal Unit Alignment
- Error Detection
- Error Correction
- Initial Alignment
- Signaling Link Error Monitoring
- Flow Control

As specified by Q.703, the functional blocks and their interaction, which make up the MTP level 2, are depicted in Fig. 2.4. The Link State Control (LSC) block together with the Initial Alignment Control (IAC) controls the overall state and mode of operation of the link including the initial alignment. The Transmission Control and Reception Control blocks take care of the transmission of signaling units (LISU, MSU or FISUs) across the link. The Delimitation, Alignment and Error Detection blocks take care of inserting flags before transmission and checking on the accuracy of the received signaling units by means of CRC and octet counting. The correct or incorrect receipt of a signaling unit are also fed into error rate monitors (AERM and SUERM) which count such statistics and decide whether the link should be operational or not. In addition, Q.703 also specified the Congestion Control and the Processor Outage Control block, which recommend, respectively, the procedures to be executed at the occurrence of congestion and processor outage in the higher layers. More on these is covered in the subsequent sections.

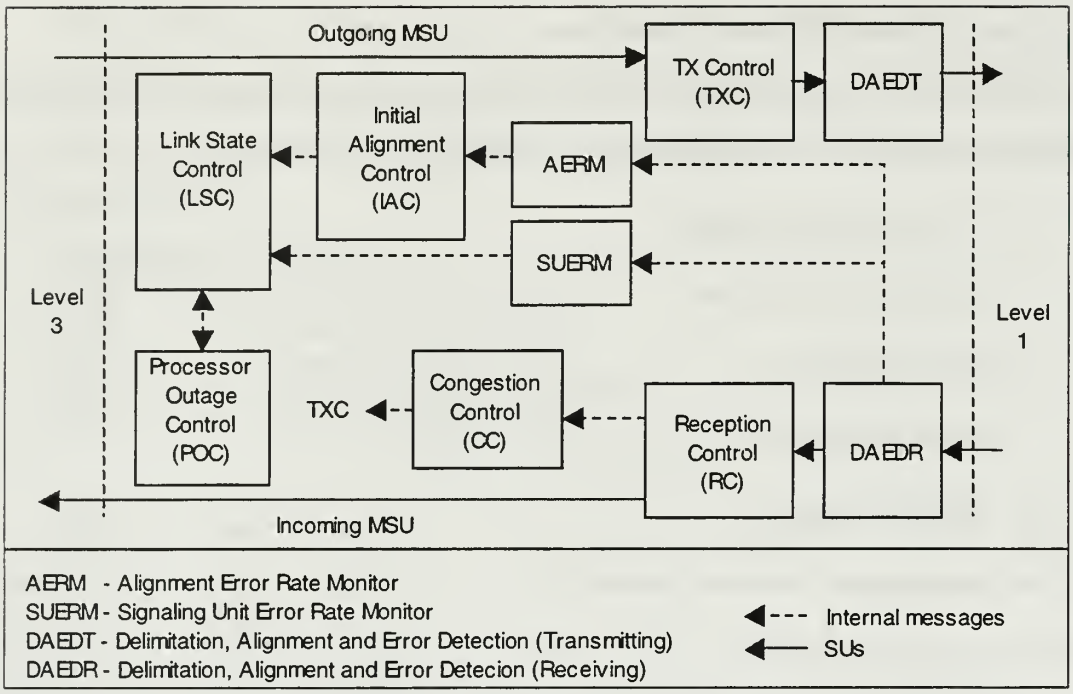

Fig 2.5 Functional Block Diagram of MTP Level 2

3. Error Detection with Flags, Zero Insertion and CRC Checks

An 8-bit Flag ('01111110') is added in front of every signaling unit before any transmission into the data link by DAEDT. DAEDT also does computation of the CRC field and zero insertion between the 5^{th} and 6^{th} bit of any continuous sequence with 6 '1's. The CRC value is generated using all the bits of the signaling unit except the Flag and it is based on the standard CCITT 16 bit CRC code.

At the remote end of the link, DAEDR checks the received SU by comparing its computed CRC with what that is received, strips off any flags, removes any inserted zeros, counts the length of signaling unit and sends it to the RC for further processing if it is all correct. Otherwise DAEDR will update the error rate monitors when it detects an error.

4. Error Correction with Go Back-N Retransmission Algorithm

SS7 employs a popular Go Back-N method to retransmit signal units that are erroneously received. When a signal unit is ready for transmission, the signaling point increments the FSN (forward sequence number) by 1 (FSN = 0..127). The CRC (cyclic redundancy check) checksum value is calculated and appended to the forward message.

Upon receiving the message, the remote signaling point checks the CRC and copies the value of the FSN into the BSN of the next available message scheduled for transmission back to the initiating signaling point. If the CRC is correct, the backward message is transmitted. If the CRC is incorrect, the remote signaling point indicates negative acknowledgment by toggling the BIB prior to sending the backward message. When the originating signaling point receives a negative acknowledgment, it retransmits all forward messages, beginning with the corrupted message, with the FIB toggled. Fig 2.5 provides an illustration of the method with SSP-A trying to send to SSP-B MSUs numbered from 5 to 9. MSU 8 is in error.

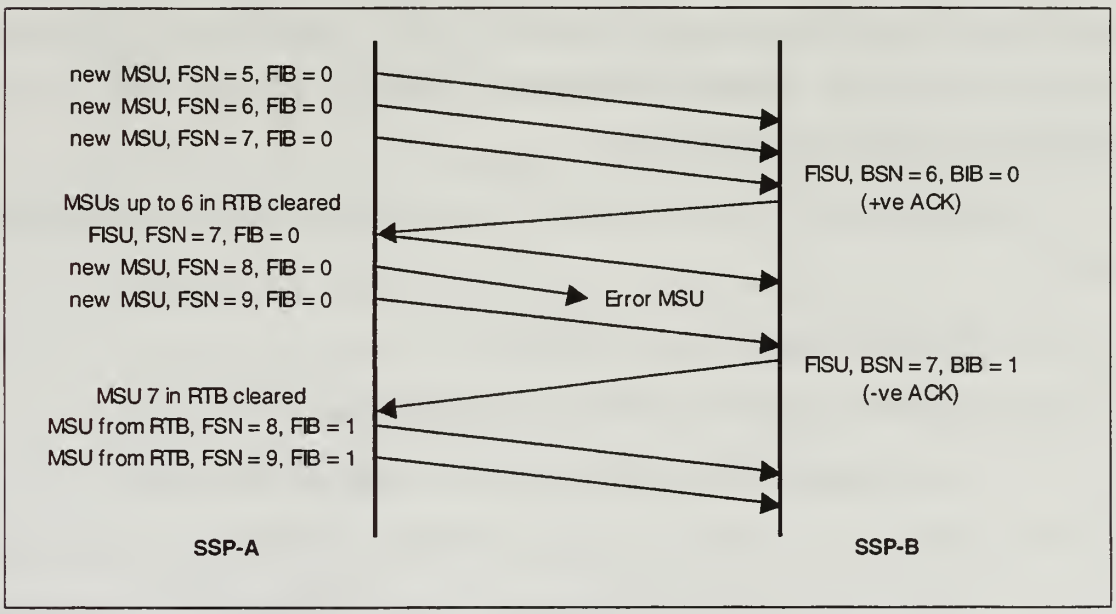
new MSU, FSN = 5, FIB = 0
new MSU, FSN = 6, FIB = 0
new MSU, FSN = 7, FIB = 0
FISU, BSN = 6, BIB = 0
(+ve ACK)
MSUs up to 6 in RTB cleared
FISU, FSN = 7, FIB = 0
new MSU, FSN = 8, FIB = 0
new MSU, FSN = 9, FIB = 0
Error MSU
FISU, BSN = 7, BIB = 1
(-ve ACK)
MSU 7 in RTB cleared
MSU from RTB, FSN = 8, FIB = 1
MSU from RTB, FSN = 9, FIB = 1
SSP-A
SSP-B

Fig 2.6 Use of Sequence Numbers and Indicator Bits in SS7

Because the 7-bit FSN can store values between zero and 127, a signaling point can send up to 128 signal units before requiring acknowledgment from the remote signaling point. The BSN indicates the last in-sequence signal unit received correctly by the remote signaling point. The BSN acknowledges all previously received signal units as well. For example, if SSP-A receives a signal unit with BSN = 3 followed by another with BSN = 6 (and the BIB is not toggled), it implies successful receipt of signal units 3 through 6 as well. It should also be noted that sequence numbers are used to identify

MSUs only. Sequence numbers are incremented when new MSUs are transmitted but kept at the previous value when LISUs and FISUs are transmitted.

5. Transmission (TXC) and Reception Control (RC)

As explained, SS7 uses the Go Back-N algorithm as the foundation to define two different types of transmission/reception methods to correct any transmission errors. For non-intercontinental or non-satellite links with one way propagation delay less than 15 ms, the Basic Transmission/Reception method is recommended, else the Preventive Cyclic Retransmission (PCR) method is recommended. Both methods keep a copy of the last transmitted MSU in a Retransmission Buffer (RTB) and only those MSUs that have been positively acknowledged are cleared (the Go-Back-N algorithm). However retransmission of the MSUs is executed whenever there is a negative acknowledgement in the Basic method but cyclically retransmitted over the link whenever there isn't any new MSUs or LISUs to send in the latter.

The basic functions of the Transmission Control block as outlined by Q.703 are as follows:

- Receive link status from LSC and IAC
- Receive and queue up new MSUs from MTP level 3
- Receive transmission inhibit/un-inhibit commands for MSUs and LISUs
- Receive BSN and BIB values from MSU received by RC and compare them with own FSN and FIB. Clear positively acknowledged MSUs or retransmit negative acknowledged MSUs
- Transmit signaling units based on the order listed in Table 2.4 with BSN and BIB fields set to represent positive or negative acknowledgement for MSUs received by RC
- Operate Timers T6 and T7 and declare link failure if they are expired

Order No.	Type of Signaling Unit
1	Any LISUs unless it is inhibited
2	Any MSU required for retransmission (Depending whether it is using Basic or PCR error correction method)
3	New MSUs unless it is inhibited or the Retransmission Buffer is full

	(A full retransmission buffer means transmission of new MSU is not possible since a copy cannot be added into the buffer)
4	FISUs

Table 2.4 Signaling Unit Transmission Order

Timer	Description
T1: Aligned ready	The time by which the link has to be in service after alignment if ready.
T2: Not aligned	The time by which the link has to be aligned ready after start of alignment.
T3: Aligned	The time by which the link has to be able to enter proving after link is aligned.
T4: Proving Period	Monitoring period for AERM.
T5: Sending SIB	The time cycle which the SIB message has to be repeatedly transmitted during a local congestion.
T6: Remote Congestion	The time by which a remote SP has to recover from congestion else link failure is declared.
T7: Excessive delay in ACK	The time by which a MSU must be acknowledged else link failure is declared.

Table 2.5 MTP Level 2 Timers

The basic functions of the Reception Control block as outlined by Q.703 are as follows:

- Receive MSU/FISU Accept and Reject commands from LSC.
- Receive and process of signaling units from remote SP. Update LSC, IAC and TXC on link status from LISUs. Check for validity of BSN and BIB fields and update TXC. Forward any accepted MSUs to MTP Level 3.
- Check the FSN and FIB fields to determine type of acknowledgement TXC should transmit. Out of sequence numbering for FSN indicates a negative acknowledgment.

6. Error Rate Monitoring

Error rate monitoring is used by SS7 to monitor the "health" of a signaling link. It is performed both when the link is in service and during the link's proving period at startup. The Signal Unit Error Rate Monitor (SUERM) performs the in-service monitoring, while the Alignment Error Rate Monitor (AERM) is employed during proving. Fig 2.7 illustrates the use of error counting procedure performed by both functions.

AERM receives indication from IAC the period within which it is to monitor the error rate. When started, AERM simply counts the number of error SUs. If the counter C_a, reaches the value T_i, the link is declare to have failed the proving period. The value of T_i varies from T_{in} to T_{ie} when the link is in normal alignment mode and emergency alignment mode respectively. Recommended values for T_i values are listed in Table 2.6.

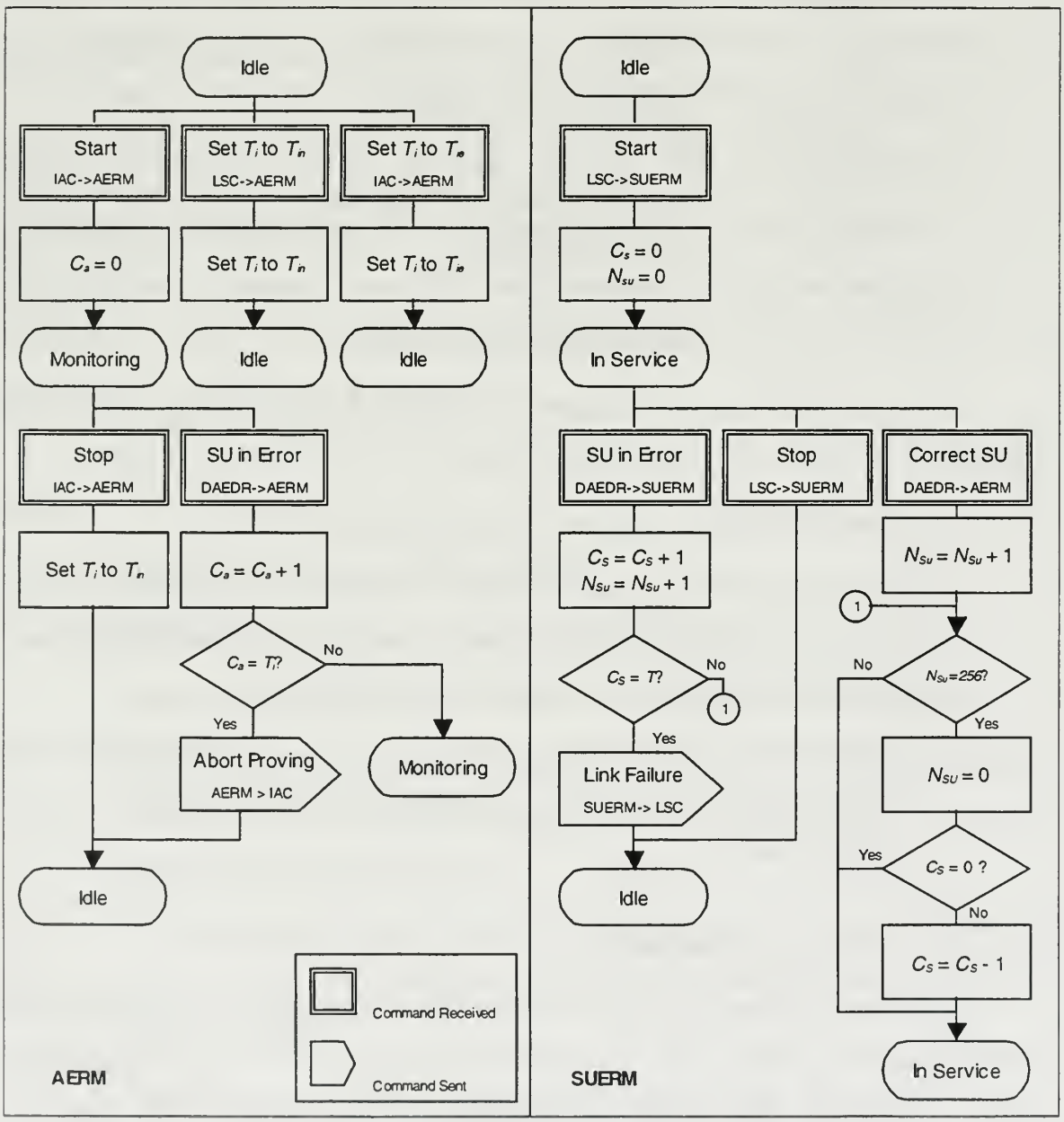

Fig 2.7 Functional Procedure of AERM and SUERM

SUERM on the other hand, operates slightly differently. AERM ensures that the link operates with a low level of error during the short proving period to determine

successful link startup and initiate MSU transmission operation. SUERM, on the other hand is designed to continuously check the "health" of the link when it is in operation. As such, besides counting error SUs, the monitor also gives credit for correctly received data. The monitor has a similar counter C_s, which counts the number of error SU received and will declare link failure if the limit T is reached. However the counter can be decremented (credited) by 1, for every 256 SU received (error or not). Recommended values of T are also listed in Table 2.6.

Ti	Data Rate	T	Data Rate
Tin = 4 Tie = 1	< 64 kbps < 64 kbps	64	< 64 kps

Table 2.6 Recommended Parameters for AERM and SUERM

7. Overall Link State Control and Initial Alignment

The overall mode of operation of MTP-2 is managed by the LSC. Prior to normal operation, the link needs to execute an initial alignment procedure, which is managed and controlled by the IAC. Upon power on, which is usually instructed by a higher level, the TXC will be controlled by the LSC to send SIOS continuously. Another link startup message would then activate the LSC and IAC to starts its alignment procedure. The TXC is directed to send SIOs and until it is reciprocated by the adjacent level and received by RC. At this point, the link is considered aligned and the TXC is directed to switch to sending SINs. However to ensure that it is stable for operation, four (4) proving cycles (usually T4 is 7.5 - 9.5 seconds for each cycle) are recommended to check for excessive error. During which the TXC will send only FISUs. Upon expiration, if AERM did not determine any excessive errors, the link will then be operational and the TXC will start transmitting any MSUs stored in its buffer. Table 2.7 illustrates this sequence of events.

Sequence of Events controlled by LSC	State of Functional Block				
	IAC	TXC	RC	AERM	SUERM
Link Power Off	Idle	Idle	Idle	Idle	Idle
-					
Power On		In Service			
TXC sends SIOS repeatedly					
Alignment Starts	Not Aligned		In Service		
TXC sends SIO repeatedly					
Link Aligned	Aligned				
SIO received by RC and TXC sends SIN					
Proving Period x 4 cycles	Proving			Monitor	
SIN received by RC					
Alignment Complete	Aligned Ready			Idle	In Service
4 successful proving period without error and TXC sends FISUs only					
In Service	Idle				
FISU receives by RC and TXC sends MSUs if any					

Table 2.7 MTP Level 2 Link Startup Procedure

E. MTP LEVEL 3

1. The MTP Level 3 Functions

The main role of MTP Level 3 [5] is to provide message routing between signaling points in the SS7 network and ensure a high availability of signaling routes. It is divided into two main functional blocks, the Signaling Message Handling (SMH) function and Signaling Network Management (SNM) function as shown in Fig 2.8.

Each MTP-2 module only takes care of transmitting/receiving MSUs for one link. As signaling points have more than one link, MTP-3 is necessary to manage the distribution of messages between links and its local users. In a complex network, MTP-3 also has to efficiently perform routing of messages across available links, towards the destination. These functions are taken care by the SMH block. MSUs from users are routed by the Routing sub-block into MTP-2 links. On the other hand, MSUs received from links are checked by the Discrimination sub-block. If they are destined for a local

user, they will be distributed to the respective user layer. Otherwise they will be sent to the Routing function for forwarding to the next signaling point.

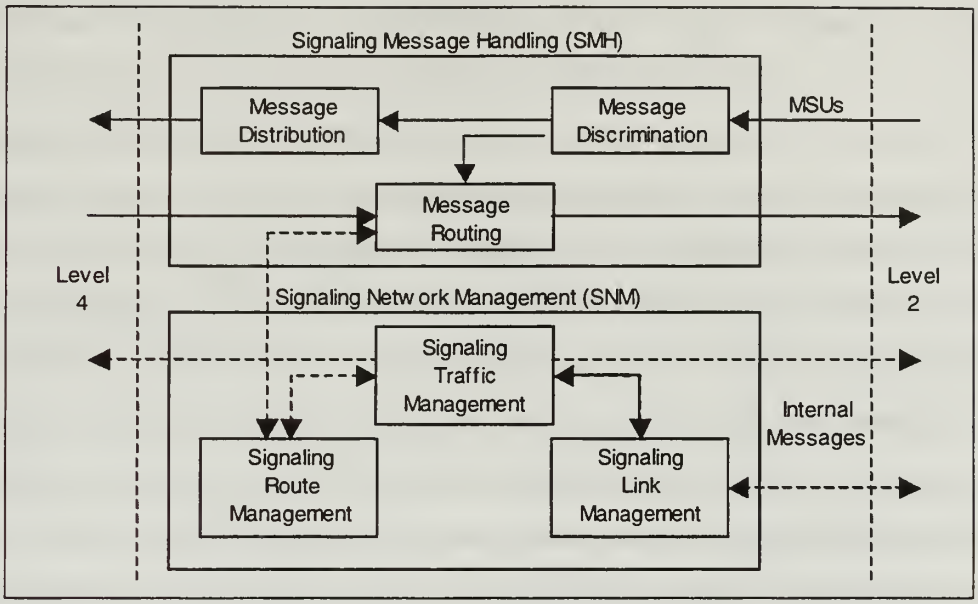

Fig 2.8 Functional Block Diagram of MTP Level 3

The provision of Signaling Network Management functions is another example of achieving high availability and reliability in SS7. This group of functions can actually be classified as a SS7 user as they make use of message transfer services to transfer network management MSUs across the network. Within a signaling point, the Signaling Link Management sub-block communicates, controls and monitors the status of MTP level 2 links connected. It provides the Traffic Management function, the overall controller, the "health" of all the connected links. This information is then conveyed to the Route Management function, which maintains the signaling points routing tables.

As a whole, the basic network management objectives are:

- Activation and deactivation of links to maintain a minimum operational network.
- Monitor link and route status to update the Routing Table.
- Reconfiguration of signaling network in case of failures and congestion to ensure minimum disruption to the network services.

2. Routing Labels

Routing of messages by MTP Level 3 is based on the routing label in the Signaling Information Field (SIF). The routing label is comprised of the Destination Point Code (DPC), Originating Point Code (OPC), and Signaling Link Selection (SLS) fields. Point codes are numeric addresses which uniquely identify each signaling point in the SS7 network. When the DPC in a message matches the point code of the receiving signaling point, the message is distributed to the appropriate user part (e.g., ISUP or SCCP) indicated by the service indicator in the SIO. Messages destined for other signaling points are routed to an outgoing link based on the DPC and SLS values.

An interesting point to note is that ANSI and ITU-T have different routing label formats as shown in Fig 2.9. ANSI point codes are 24-bits with a 5-bit SLS while ITU-T point codes typically use 14-bits and a 4-bit SLS field. For this reason, signaling information exchanged between ANSI and ITU-T networks must be routed through a gateway STP or protocol converter, which has both an ANSI and an ITU-T point code.

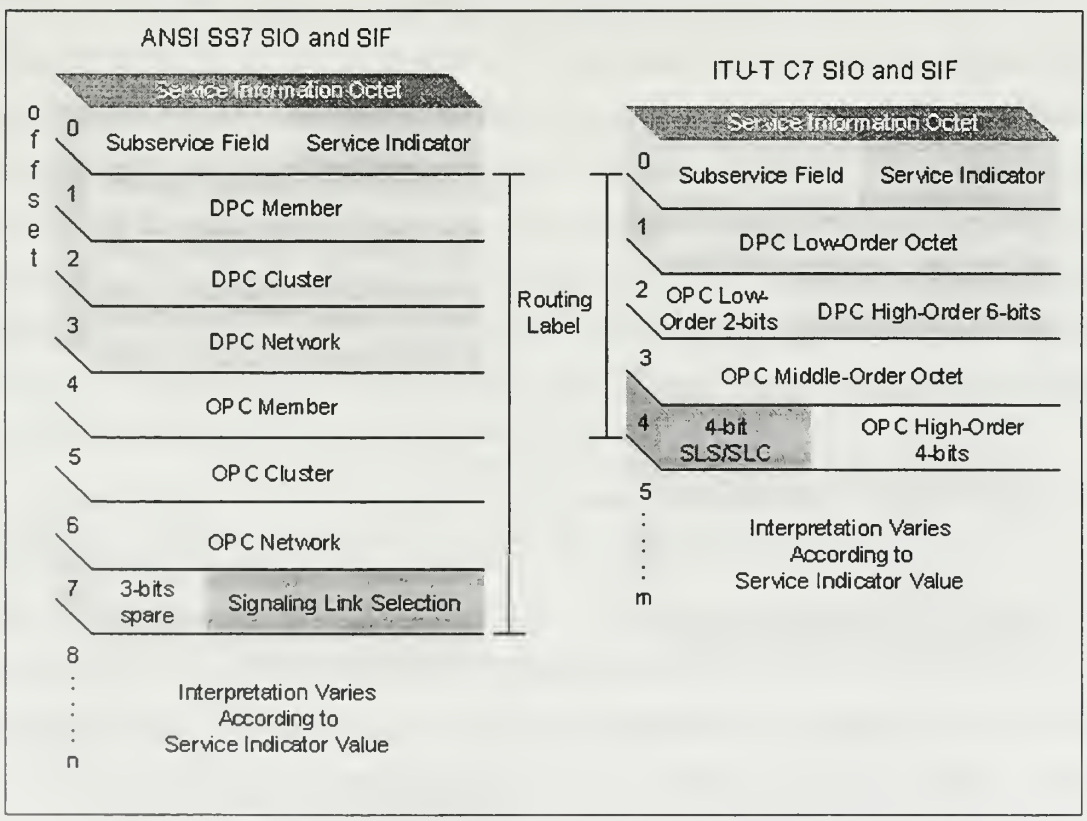

Fig 2.9 ANSI vs. ITU-T SIO and SIF Format

An ANSI point code consists of network, cluster, and member octets (e.g., 245-16-0). ITU-T point codes are pure binary numbers, which may be stated in terms of zone, area/network, and signaling point identification numbers. For example, the point code 5557 (decimal) may be stated as 2-182-5 (binary 010 10110110 101).

3. Routing of Messages

The selection of outgoing link is based on information in the DPC and SLS fields. The DPC is used to determine the possible outgoing linkset(s) and the SLS field is used to decide the outgoing link from the links among the possible linkset(s). As an example, if each linkset for a SSP (Fig 2.2) has two links, each SSP will have 4 equally possible outgoing links to reach the other SSP. If the SSPs operates using ITU-T 4-bit SLS format (giving 16 possible codes), each of the links will have 4 SLS values assigned. And in theory, if a user part sends messages at regular intervals and assigns the SLS values in a round-robin fashion (rotate from code 0 to 15 equally), the traffic level should be equal among all the links. In addition, this form of SLS routing also ensures that messages arrive with proper sequencing. Any two messages sent with the same SLS will always arrive at the destination in the same order in which they were originally sent.

However, the use of SLS does pose a problem in that even load distribution is not achievable for all link configurations. With a 4 bit SLS, a configuration with 3 links in each of the two linksets (totaling 6 links) results in an uneven assignment of 3 SLS values for 4 links and 2 SLS values for the remaining 2 links. Similar problems are encountered in the ANSI recommendation.

THIS PAGE IS INTENTIONALLY LEFT BLANK

III. OPNET MODELLING

A. GENERAL CAPABILITIES OF OPNET

OPNET is a suite of network simulation software developed by MIL3 Inc. It provides direct and easy modular implementation of communications network topologies. OPNET uses a hierarchical and object oriented approach towards network modeling. The most fundamental construction unit of a network in OPNET is a module. There are several types of modules as illustrated in Fig 3.1. Together, modules can be used to construct any network component such as a host computer, a router or hub, etc. The constructed network component is called a node in OPNET and it could then be used to construct any network or subnet as desired.

As a result, a real world network or subnet is represented by a network of nodes with each node modeled after a host, router or other component in the network. Each type of node is in turn modeled based on the modules it consist of and the processes they execute.

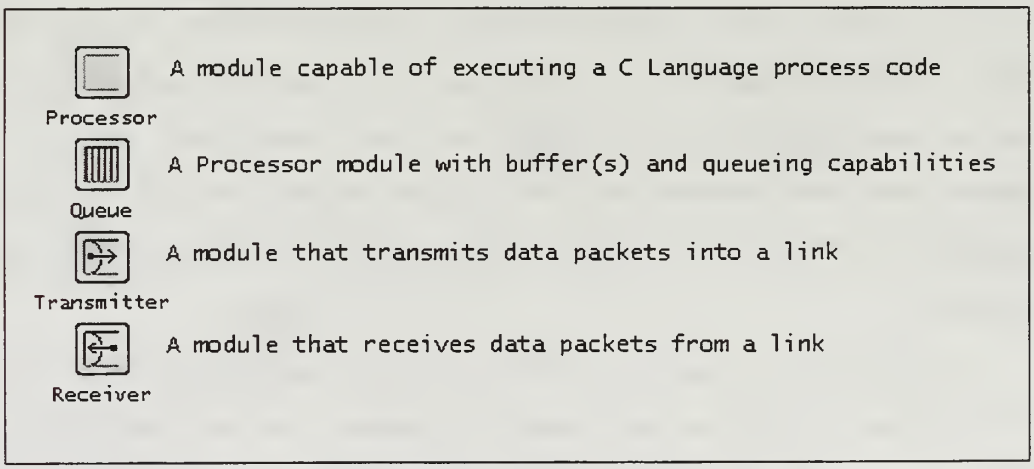
A module capable of executing a C Language process code
Processor
A Processor module with buffer(s) and queueing capabilities
Queue
A module that transmits data packets into a link
Transmitter
A module that receives data packets from a link
Receiver

Fig 3.1 Main Types of OPNET Modules

The use of OPNET in modeling the SS7 network will be discussed in the following sections. The major components of OPNET (version 6.0) used are tabulated in Table 3.1.

Software Module	Description
Project Editor	Specify network topology and configure nodes and links. Choose results, run simulations, and view results.
Node Editor	Create models of nodes by specifying internal structure and capabilities.
Process Editor	Develop models of decision-making processes representing protocols, algorithms, resource managers, operating systems, etc.
Link Editor	Create, edit, and view link models.

Table 3.1 Major OPNET Version 6.0 Components

B. MODELLING SIGNALING POINTS AND NETWORKS USING OPNET

With reference to the procedures and functions envisaged in ITU-T Q.7xx-Series Recommendation, the flow of MSUs within the Message Transfer Part could be represented with processor-buffer models. Consider a two link SSP with its model as shown in Fig 3.2. An OPNET model would comprised mainly of queue and processor modules. Logically, MTP-3 should have three queue modules, one for each function (routing, discrimination and distribution). Likewise each MTP-2 should have one queue (for TXC) and one processor module (for RC) for distinct functions. However this would make a SSP with multiple links very complex to model and make internal communication between functional blocks and between levels very complicated. Instead, a simpler approach would be to represent each level with just one queue module. There is no loss in simulation capability and our OPNET model would be smaller and more modular to construct.

To illustrate the above, the OPNET model for the two link SSP is shown in Fig 3.3. It includes a simple processor module to simulate a user part function that could be ISDN or Data user part etc. There are three queue modules, one to simulate MTP-3 and one for each MTP-2 function. The pairs of transmitter/receiver modules are necessary for OPNET to connect a processor/queue module to a link. A pair of these modules would represent one physical link. It can be seen that modular construction of SS7 signaling points with other protocol structures can be readily done in OPNET.

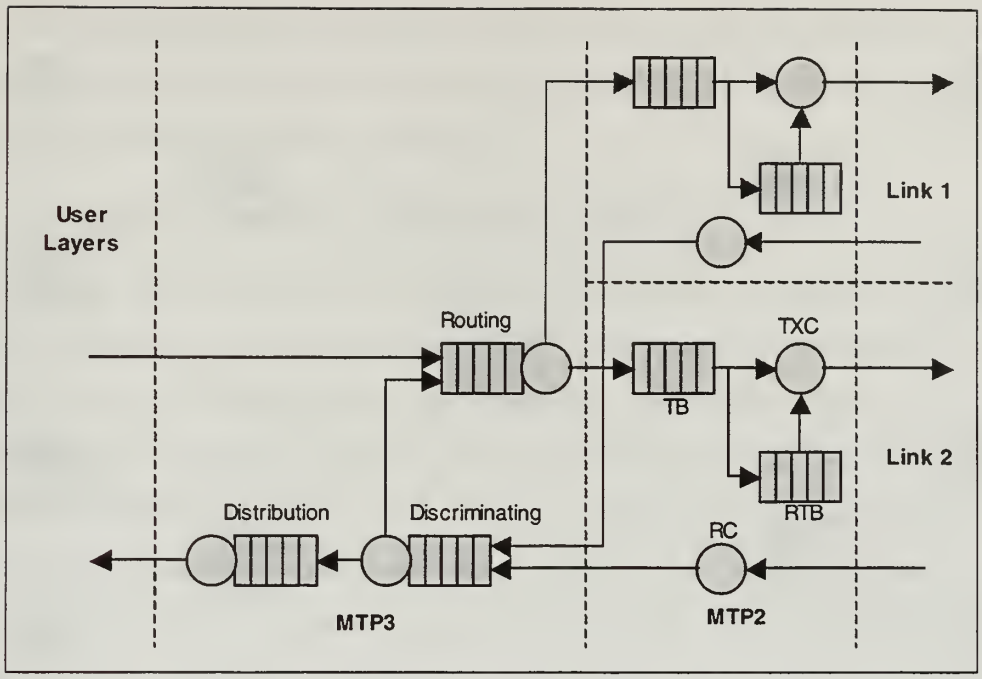

Fig 3.2 Processor-Buffer Model of an SSP with 2 Links

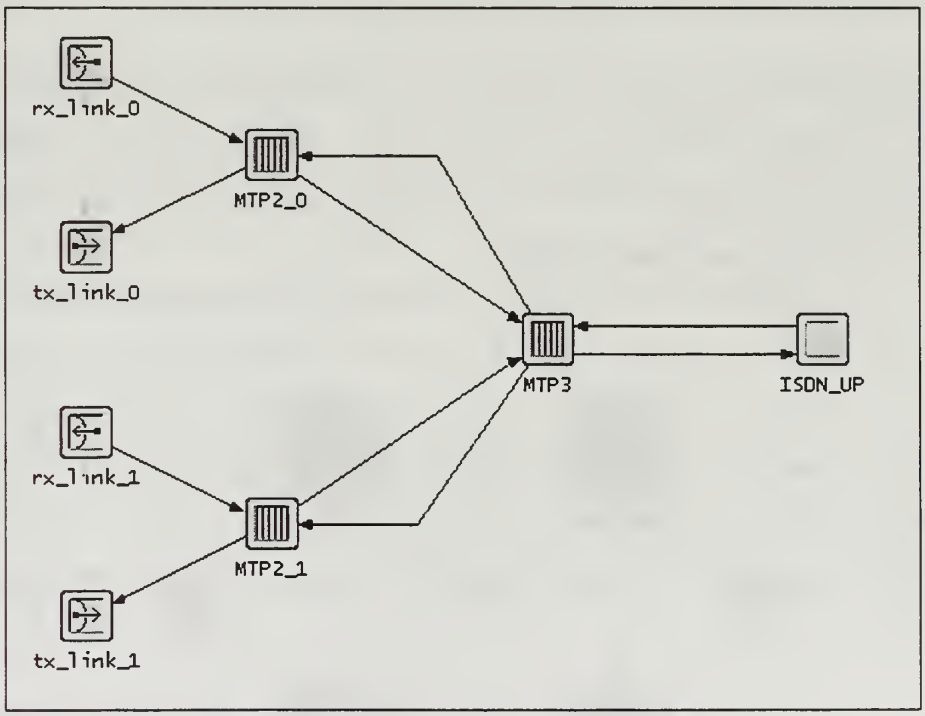

Fig 3.3 A 2 Link SSP with ISDN User Part

As another example, a STP which manages 4 SS7 links would have a protocol structure as shown in Fig 3.4. In this case, no high-level user modules are modelled as STPs generally only serve as message transfer points.

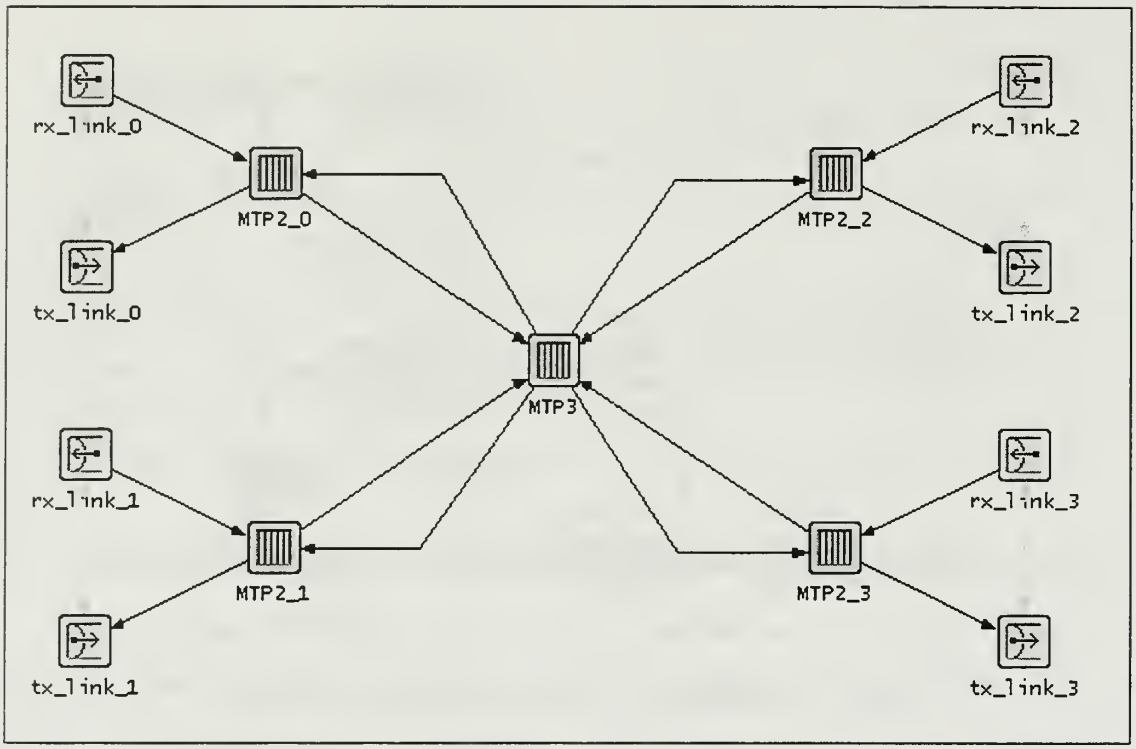

Fig 3.4 A 4 Link STP

So far, discussion is based on the model of a signaling point represented by a node in OPNET. To further illustrate the construction of a SS7 network using defined node models, a mesh network with 2 SSPs and 4 STPs that uses the node model as in Fig 3.3 and 3.4 respectively is constructed in Fig 3.5.

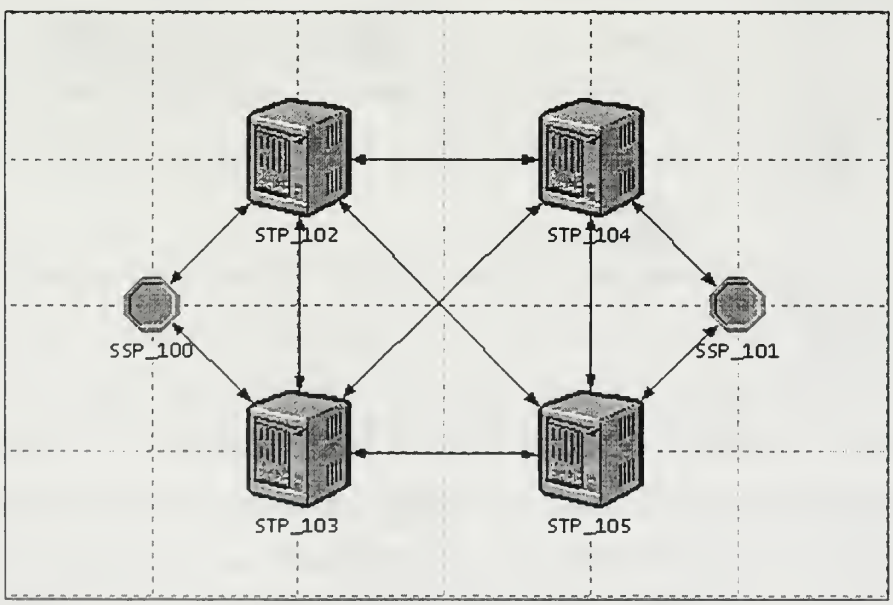

Fig 3.5 A Typical 2 SSP SS7 Network

Having discussed the construction of the various types of signaling points and network nodes and modules, the following section shall discuss the specific modeling of the MTP processes within OPNET modules.

C. MODELING MTP PROCESSES USING OPNET

OPNET allows discrete event modeling of processes. Typically, in simulating network protocols, one can just simulate the characteristics concerned and record statistics about that. For example, if the concern is transmission delay, one can just simulate the data rate, packet length and protocol method in which the data packets are transmitted and then measure the delay. However this approach produces very specific and simple process models, which are restricted in use. A need for recording other statistics would require the user to reconstruct the model all over again from top to bottom.

As such in the modeling of SS7 processes, a concerted effort was made to model as many of the protocol characteristics as possible, not just in the behavior but also according to how a real system would work. This includes exact modeling of the functions performed by the protocol. The MTP layers, being the fundamental building blocks of any SS7 network, are selected as the prime processes. The ITU-T standards, being the reference standard of SS7, are used for reference and modeling.

Of the three MTP layers, both Level 2 and 3 requires modeling. However the main work of this thesis is on the modeling of MTP-2. The modeling of MTP-3 is conducted by Ow [6]. Although both levels are modeled separately, the modular approach in the design of the signaling points architecture in OPNET allows direct integration of the two levels, thereby providing a complete SS7 Message Transfer Layer ready for simulation.

D. MODELING MTP LEVEL 2 PROCESS

1. Model Overview

The process model of MTP Level 2 is constructed based on ITU-T Q.703 standard. This means the various functional blocks as depicted in Fig 2.5 have to be

implemented. Fig 3.6 represents the OPNET model for this process with 14 states (the circular symbols). Most of the time, each state is directly assigned to implement one functional block, however, the states "TXC", "TXC_rcv" and TX_MSU" implement the combined functions of TXC and DAEDT, and the states "RC" and "RC_rcv" implements the combined functions of RC and DAEDR. The states "TXC" and "RC" handle only inter function communication requirements while the states "TXC_rcv" and "RC_rcv" handle the receiving of MSUs from MTP-3 and MTP-1 respectively. The separate state "TX_MSU" is used to implement the transmission of signaling units onto the link (MTP-1) and also allow the incorporation of transmission time. The "init" state is used for initializing the state variables while the "idle" state is the default state the process would be in whenever it has completed its task in the other states. Lastly the "ext_msg" state is used to handle inter function messages arriving from other SS7 layers.

When an OPNET simulation is executed, the "init" state will be executed and then the process will stay "idle" until an event such as arrival of a MSU or a inter-function message. For example if a link startup message from MTP-3 for LSC is sent to the level, "ext_msg" would be executed to pass the message to "LSC". "LSC" upon receiving the message would then execute its link startup procedures, which includes sending messages to "IAC" and "TXC" and so on. Thus a continuous chain of events is constantly happening in the process simulating the working of a typical SS7 layer.

2. Buffers and Processor Delay

Each OPNET queue module comes with a processor and facility to define the number of buffers associated with the processor. For MTP Level 2, the main buffers would be the MSU transmission and Retransmission buffer. However based on Q.703 specifications, a practical implementation of the protocol would also require buffers for the inter-function messages and commands. In fact, every functional block should have its own message buffer (Table 3.2) from which it could extract a message intended for itself and then act on it. Such a mechanism would also allow incorporation of processor delay (the message service time) into the model. In our OPNET model, inter-function message processing for each function is assumed to have a constant service rate. The service rate of MSUs from the transmission and retransmission buffer is based on its

packet length and link data rate. All buffers are also designed to have infinite size since generic Q.703 does not incorporate any procedure for handling buffer overflow. Of course, in the practical world this is not possible, nevertheless most simulation analyses assume a large buffer in the system. And in the event buffer overflow conditions are to be studied, redefinition of the buffer size and incorporation of an overflow handling procedure can still be done in OPNET readily.

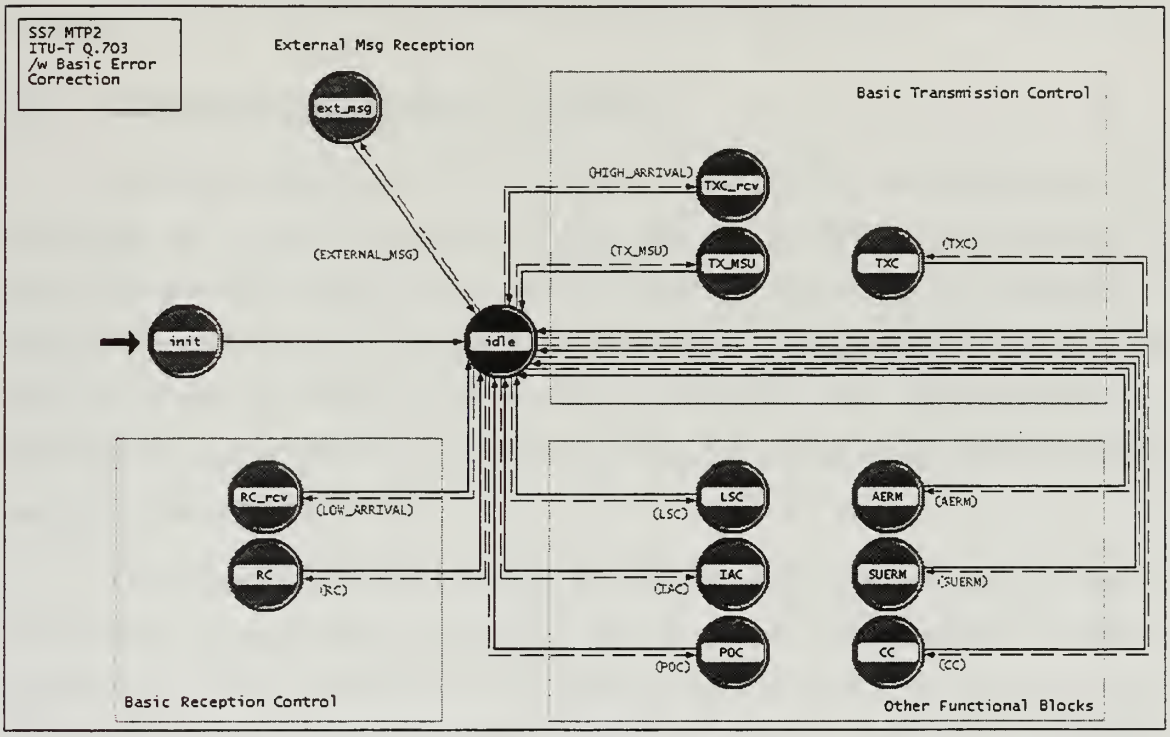

Fig 3.6 Process Model of MTP Level 2

Buffer Id	Function	Service Rate
0	Transmission buffer	Data rate and MSU length dependent
1	Retransmission buffer	
2	LSC internal message buffer	Constant
3	IAC internal message buffer	(can be defined prior simulation)
4	POC internal message buffer	
5	TXC internal message buffer	
6	RC internal message buffer	
7	AERM internal message buffer	
8	SUERM internal message buffer	
9	CC internal message buffer	

Table 3.2 MTP Level 2 buffers

E. MODELLING OF MESSAGE TRANSFER PART LEVEL 3

As mentioned, the modeling of MTP Level 3 using OPNET is performed by Ow [6]. The approach is similar to that for MTP Level 2. The whole layer is implemented using one OPNET queue module. The functional blocks of the protocol are implemented as processing states in OPNET and buffers are allocated for storing MSUs and inter-function messages. Further elaboration of the model is discussed in the reference.

IV. LINKSET DELAY USING QUEUE MODELING

A. INTRODUCTION

This chapter outlines the mathematical derivations of the Average Linkset delay established by Wong [7] using Markov theory. The predictions obtained are then compared with a simulation run using the constructed OPNET Model from the previous chapter.

B. LOAD DISTRIBUTION IN A LINKSET

The SS7 protocol uses the SLS code for the purpose of load sharing. The originating user of an MSU generates a SLS code and the MTP layers determine on which link the MSU should be sent. The SLS codes are assumed to be generated in a round robin fashion by the user layer. The routing table at a signaling point includes all the possible links that could be used to reach the destination. For each possible link, an SLS code is assigned. MSUs arriving at the Signaling Point are then sent via the link that matches the assigned SLS code.

As explained in Chapter II, Section E.2, load sharing is equal for all links only if the number of possible links is a power of 2 (i.e., 2, 4, 8 etc). For example if we have 5 possible links, with a 4 bit SLS (i.e., $N = 16$ SLS codes), 4 of the links will have 3 SLS codes assigned while the 5th link will have 4. Thus more traffic will flow into the 5th link if SLS codes are uniformly generated. In general, if there are K possible links for a destination, and λ_{LS} is the total traffic load in MSUs per sec, there will be n_H links with traffic λ_H and n_L link with traffic λ_L.

$$n_H = \begin{cases} K & \text{if } \mathrm{mod}(N,K) = 0 \\ \mathrm{mod}(N,K) & \text{otherwise} \end{cases} \quad (1)$$

$$n_L = K - n_H \quad (2)$$

$$\lambda_H = \frac{\left\lceil \frac{N}{K} \right\rceil}{N} \cdot \lambda_{LS} \quad (3)$$

$$\lambda_L = \begin{cases} 0 & \text{if } \mathrm{mod}(N,K) = 0 \\ \dfrac{\left\lceil \dfrac{N}{K} \right\rceil - 1}{N} & \text{otherwise} \end{cases} \quad (4)$$

The following tabulates values of n_H, n_L, λ_H and λ_L for 4 bit SLS.

K	n_H	n_L	λ_H/λ	λ_L/λ
1	1	0	1	0
2	1	1	1/2	½
3	1	2	3/8	5/16
4	4	0	1/4	0

Table 4.1 n_H, n_L, λ_H and λ_L for 4 bit SLS

C. LINK SET DELAY MODELLING

1. The M/G/1 Queuing Model

Wong [7] modeled each link in a linkset as a single stage M/G/1 queue as shown in Fig 4.1. The traffic load in each link will be equal to either λ_H or λ_L as derived in Eqn. (3) and (4) respectively. This means that the arrival process, i.e., the arrival of MSUs to the MTP layers, is Poisson and each link is processed by a single server with a General distribution in service time.

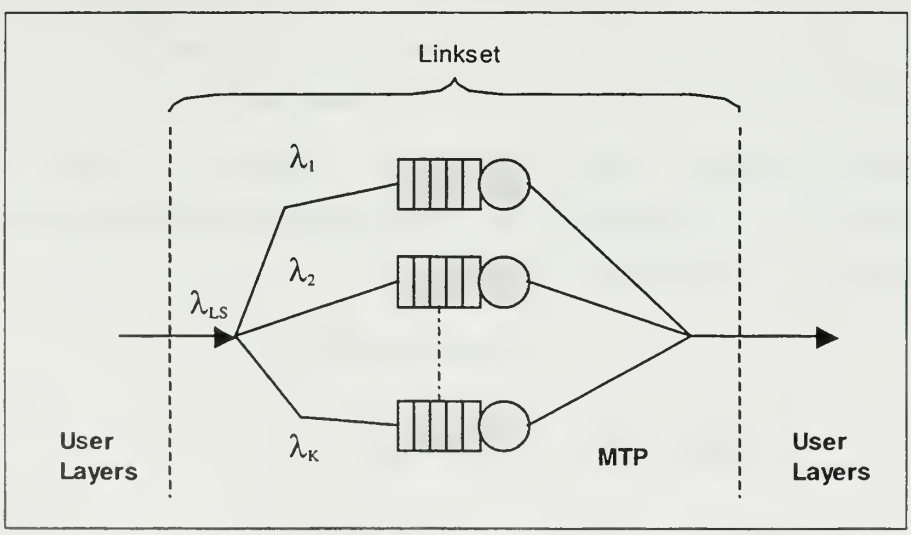

Fig 4.1 Single Stage Queuing Model for Link Set

2. General Distribution for Service Time

The distribution of the MSU service time depends on the link capacity (a constant) and the message length distribution. Let x be a random variable denoting the message length in bits with $\bar{x}$ being the average and C_x the coefficient of variation as defined in Eqn. (5) and (6). Symbol p_k, represents the probability of having a MSU with length x_k.

$$\bar{x} = \sum p_k \cdot x_k \tag{5}$$

$$C_x = \frac{\sqrt{\sum p_k \cdot x_k^2 - \bar{x}^2}}{\bar{x}} \tag{6}$$

The values of C_x are between 0 and 1. If C_x = 1, the message length is exponentially distributed, and since link capacity is constant, the service time τ, will also exponential and we would have a M/M/1 queue.

If we let the link capacity be C bits/s, using Pollaczek-KhinChin formula [8] the average service time (i.e. the waiting time to be served), $\bar{T}_s$ is defined as in Eqn. (7).

$$\bar{T}_s = \frac{\lambda \cdot E[\tau^2]}{2 \cdot (1 - E[\tau])} = \frac{\lambda \cdot \sum p_k \cdot \frac{x_k^2}{C^2}}{2 \cdot \left(1 - \lambda \cdot \frac{\bar{x}}{C}\right)} \tag{7}$$

We can also determine the link utilization ρ.

$$\rho = \frac{\text{Link Traffic}}{\text{Link Capacity}} = \frac{\lambda \cdot \bar{x}}{C} \tag{8}$$

Further substitution of C_x and ρ into Eqn. (7) and (8) respectively yields the following equations.

$$\bar{T}_s = \frac{\lambda \cdot \bar{x}^2 \cdot \left(1 + C_x^2\right)}{2 \cdot C^2 \cdot (1 - \lambda \cdot \frac{\bar{x}}{C})} \tag{9}$$

$$\bar{T}_s = \frac{\bar{x} \cdot \rho \cdot \left(1 + C_x^2\right)}{2 \cdot C \cdot (1 - \rho)} \tag{10}$$

3. Average Link Delay Estimation

For a SS7 link, the average link delay T_d is the sum of the transmission time T_t, the service time T_s and the propagation time T_p. The propagation T_p varies and it depends on the path length L and the speed v of the medium.

$$T_d = T_t + T_s + T_p \tag{11}$$

$$T_d = \frac{\bar{x}}{C} + \frac{\bar{x} \cdot \rho \cdot \left(1 + C_x^2\right)}{2 \cdot C \cdot (1 - \rho)} + \frac{L}{v} \tag{12}$$

4. Average Linkset Delay Estimation

After knowing the average delay in a link, we are now ready to determine the average delay in a link set. Assuming that there are K links in the link set, then the traffic load in each link shall be λ_i, where $i = 1...K$. Therefore using Little's Formula [8], we can determine the average number of MSUs, $\overline{N}$ and average delay, T_{LS} for the link set.

$$\overline{N} = \sum_{i=1}^{K} \lambda_i \cdot T_{d,i} = n_H \cdot \lambda_H \cdot T_H + n_L \cdot \lambda_L \cdot T_L \tag{13}$$

$$T_{LS} = \frac{\overline{N}}{\sum_{i=1}^{K} \lambda_i} = \frac{n_H \cdot \lambda_H \cdot T_H + n_L \cdot \lambda_L \cdot T_L}{\lambda_{LS}} \tag{14}$$

Where T_H and T_L can be obtained by direct substitution of the constants λ_H and λ_L respectively into Eqn. (12) to yield Eqn. (15).

$$T_{LS} = \begin{array}{l} n_H \cdot \dfrac{\lambda_H}{\lambda_{LS}} \cdot \left(\dfrac{\bar{x}}{C} + \dfrac{\lambda_H \cdot \bar{x}^2 \cdot \left(1 + C_x^2\right)}{2 \cdot C^2 \cdot (1 - \lambda_H \cdot \frac{\bar{x}}{C})} + \dfrac{L}{v} \right) + \ldots \\ n_L \cdot \dfrac{\lambda_L}{\lambda_{LS}} \cdot \left(\dfrac{\bar{x}}{C} + \dfrac{\lambda_L \cdot \bar{x}^2 \cdot \left(1 + C_x^2\right)}{2 \cdot C^2 \cdot (1 - \lambda_L \cdot \frac{\bar{x}}{C})} + \dfrac{L}{v} \right) \end{array} \tag{15}$$

A plot of the Average Linkset delay T_{LS} (normalized to the sum of T_t and T_p) against various Total Linkset Traffic Intensity λ_{LS}, is provided in Fig 4.2.

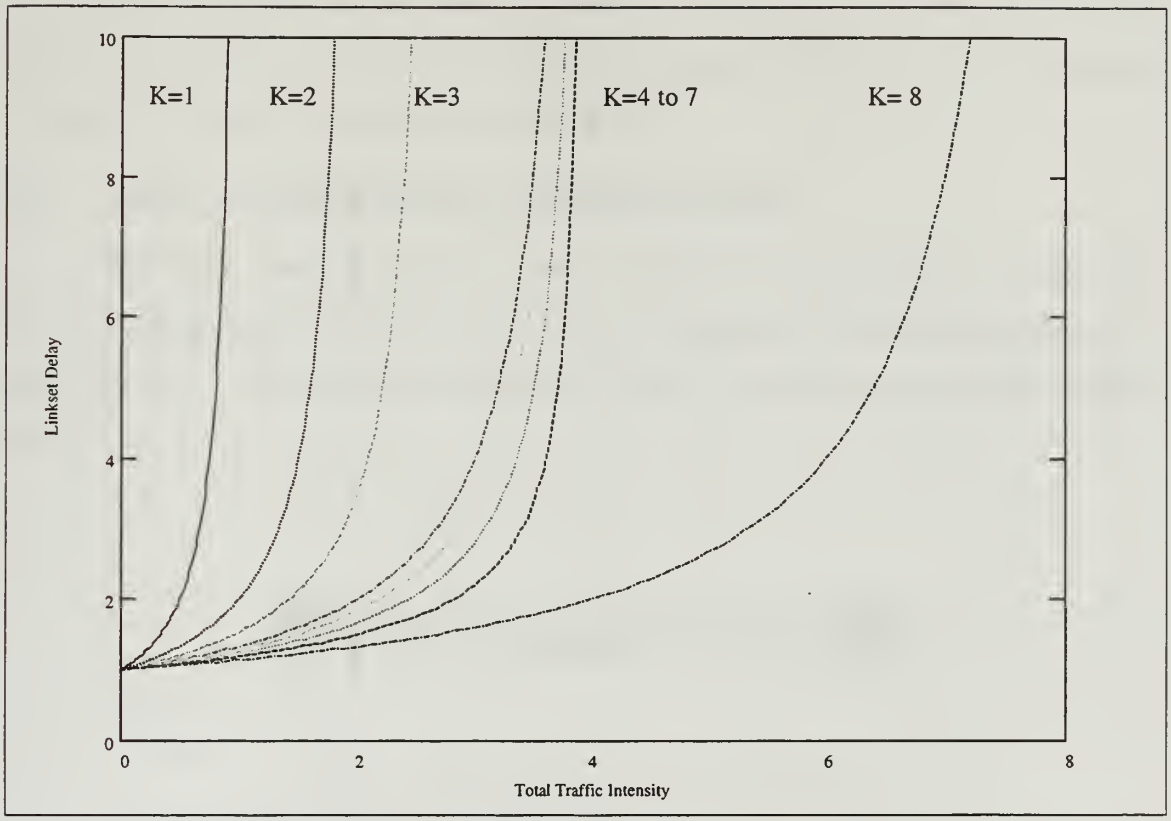

Fig 4.2 Queuing Theory Prediction of Link Set Delay (for 4 bit SLS)

THIS PAGE IS INTENTIONALLY LEFT BLANK

V. SIMULATION RESULTS AND COMPARSION

A. SCOPE OF SIMULATION

In this chapter, the simulation of a simple two SSP network will be discussed. The focus of the simulation will be to determine the average linkset delay in the network and compare it with the analysis provided in Chapter IV.

B. SIMULATION NETWORK CONFIGURATION

The SS7 network used for the simulation is comprised of two SSPs communicating between each other using two 56 kbps links. The network structure is as shown in Fig 5.1 and the internal structure of the SSP node is similar to that described in Fig 3.3.

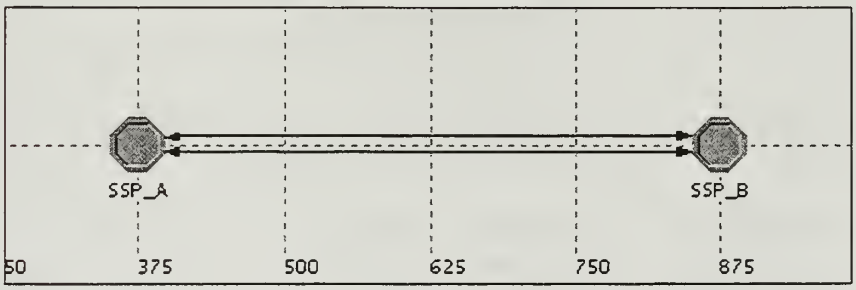

Fig 5.1 2 SSP Network Used for Simulation

In the simulation, MSUs with exponentially distributed length, are transmitted into the message transfer layer at an exponentially distributed time interval. This constitutes a Poisson arrival with exponentially distributed service time. The SLS field in the MSUs is also assigned in a round robin fashion such that both links have about the same load. For every MSU, the duration between the transmission and reception by the user part in the other SSP is recorded. These records are then used to determine the average Linkset Delay.

An important point to note is that, the average Linkset Delay equations derived in Chapter IV are based on a single stage M/G/1 queue, whereas the proposed simulation model consists of several queuing stages as shown in Fig 3.2. Further, the mathematical model only represents the transmission process in MTP Level 2 but not the routing process in MTP Level 3. The Linkset Delay obtained from the simulation would render

model inadequate, if delay in MTP Level 3 is significant or if it exhibits different characteristics. Consequently, two simulation runs were executed. The first was performed with the processing time in the MTP level 3 reduced to a very small value such that it is insignificant in the simulation. The second, on the other hand allows the processing time in the MTP level 3 to be of the same order as in MTP Level 2. The parameters used are summarized as follows:

Item	Details
Link	- Two bi-directional Links between two SSP, $K = 2$ - Data rate per link, $C = 56$ kbps - Total data rate, $2C = 112$ kbps
MSU Length	- Exponentially distributed - Average Length, $\bar{x}$ = 33 bytes
Traffic Load	- Exponential inter-arrival - Average time interval varies from 0.0024 to 0.05 sec (This generates on average: 417 MSU/s to 20 MSU/s or 110 kbps to 5.28 kbps or 98.2% to 4.7%)
SLS	- 4 bit SLS (16 SLS codes, 8 for each link) - Assigned in round robin fashion
Propagation	- Path length, $L = 1000$ km - Link speed, $v = 2 \times 10^8$ m/s
MTP-3 Service Rate	- Constant - 0.1 ms or 10,000 MSU/s used for simulation one - 1, 2, 3 and 4 ms or 1000, 500, 333 and 250 MSU/s used for simulation two
Min. Delay	$\frac{\bar{x}}{C} + \frac{L}{v} = 7.2$ ms

Table 5.1 Simulation Parameters

C. LINKSET DELAY COMPARISON

The Average Linkset Delay obtained from the first simulation is plotted against the mathematical predictions (with $K = 2$) in Fig. 5.2. Traffic loads are normalized to the data rate of one link, i.e. 56 kbps. The plot only shows the simulation data obtained with the service rate for MTP Level 3 processors set at 0.1 ms. This value is chosen such that the MTP-3 layer is insignificant compared to the overall minimum possible delay of 7.2 ms. This allows us to compare our OPNET model for MTP-2 with the single stage M/G/1

queuing model. The simulation results show that the two models are quite similar yielding only slight differences. The simulation generates a slightly larger delay for all traffic loads. This can be explained by the fact that the theoretical model assumes no processor delay in MTP-3. In reality, an outgoing MSU would have to pass the routing function of MTP-3 prior passing through the MTP-2 layer and onto the link. At the receiving end, the MSU has to be processed by the Discrimination function and Distribution Function of MTP-3 too. All of this additional processing adds to the overall delay.

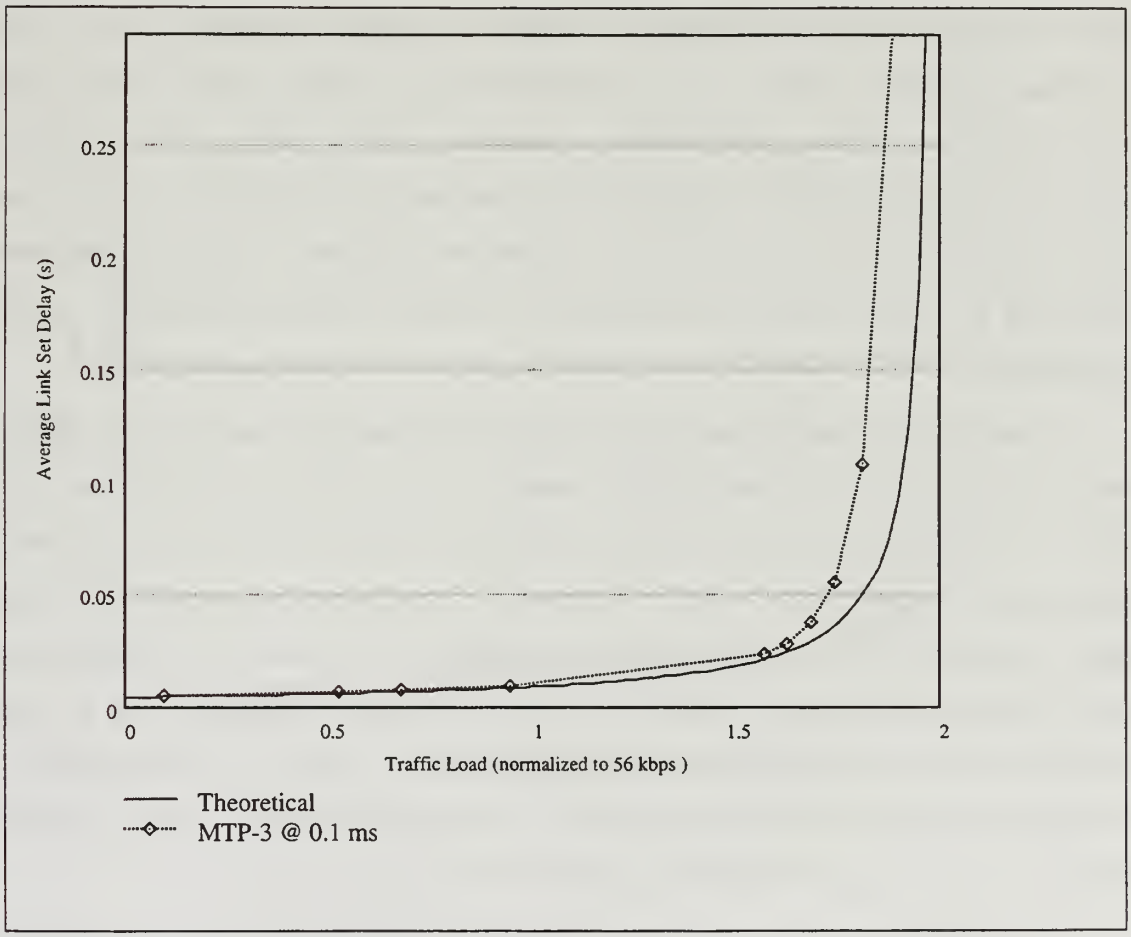

Fig 5.2 Average Link Delay Comparison Between Simulation and Estimation

Subsequently, the MTP-3 processors' service rate is increased to more significant values at 1 , 2, 3 and 4 ms. The average linkset delay obtained from simulation at these rates are plotted in Fig 5.3. It can be observed that there is not much difference in the average linkset delay when MTP-3 service rate is either 0.1, 1 or 2 ms. However the

delay characteristic changes completely and increases significantly when MTP-3 service rates are at 3 and 4 ms. The increase is also more significant at the slower 4 ms rate than the 3 ms rate. A check on the MTP-3's queuing delay reveals that there is congestion in the routing function at the 3 and 4 ms rates but not the earlier. The other functions of MTP-3, the Discrimination Function and Distribution Function, do not contribute much delay.

At a total linkset capacity of 112 kbps, the highest possible MSU generation rate for our data is 424.4 MSU/s (i.e. on average one MSU per 2.36 ms interval). When the MTP-3 service rates are faster than this value, the effect on average linkset delay contributed by MTP-3 remains small and invariant since the routing function in the layer can still handle the arriving MSUs. Thus we observed that the average linkset delay is similar to the single stage M/G/1 model with service rates at 0.1, 1 and 2 ms. Fig. 5.4 and Fig 5.5 shows that MTP-3 can still serve its function effectively at close to maximum traffic load 417 MSU/s (or 1.96 normalized to 56 kbps). The overall delay is mostly contributed by MTP-2 as the MSUs stay much longer in the layer instead of MTP-3.

However this is not the case at the slower service rates. At 3 and 4 ms, the MTP-3 routing function could only handle traffic loads up to 1.57 and 1.18 respectively (i.e. MSU generation interval of 3 and 4 ms respectively). Beyond that, the average linkset delay becomes exponentially large. In other words, the linkset becomes MTP-3 layer limited instead of link data rate limited when traffic load exceeds what MTP-3 could handle. The linkset delay is not stable but instead, it increases linearly in time as more and more MSUs are queued but not served in the Routing Function. Fig 5.6 and Fig 5.7 shows that the MSUs are increasingly stuck in the routing function's buffer at a traffic load of 1.63 (i.e. average MSU generation interval of 2.9 ms).

Thus, we can conclude that Wong's [7] single stage model can only provide good predictions to average Linkset Delay if the processing delay in the MTP-3 layer is not significant compared to the traffic load. The service rate of the MTP-3 routing function must be higher than the average MSU generation rate. Otherwise the single stage model would not be adequate since it is not capable of predicting the accumulation of MSUs in the routing function observed using simulation.

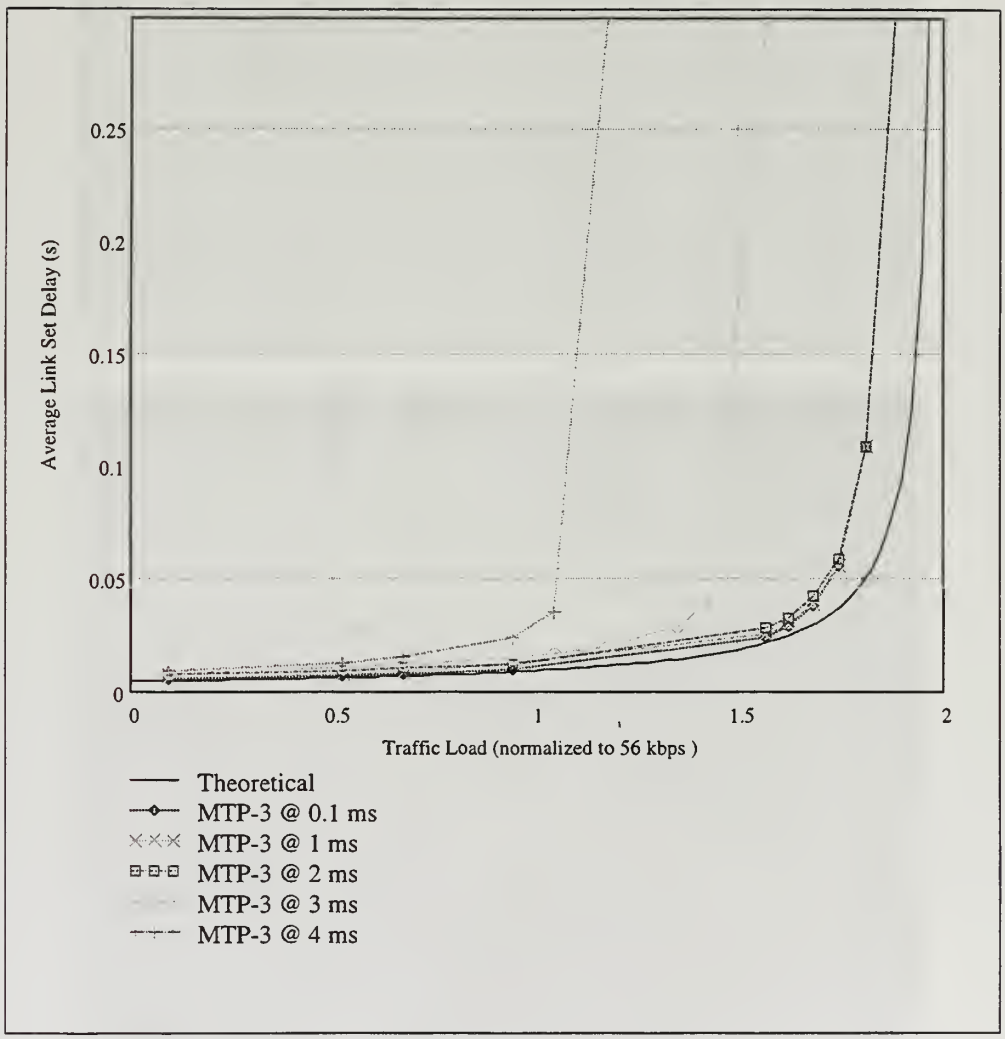

Fig 5.3 Average Linset Delay at Various MTP-3 Service Interval

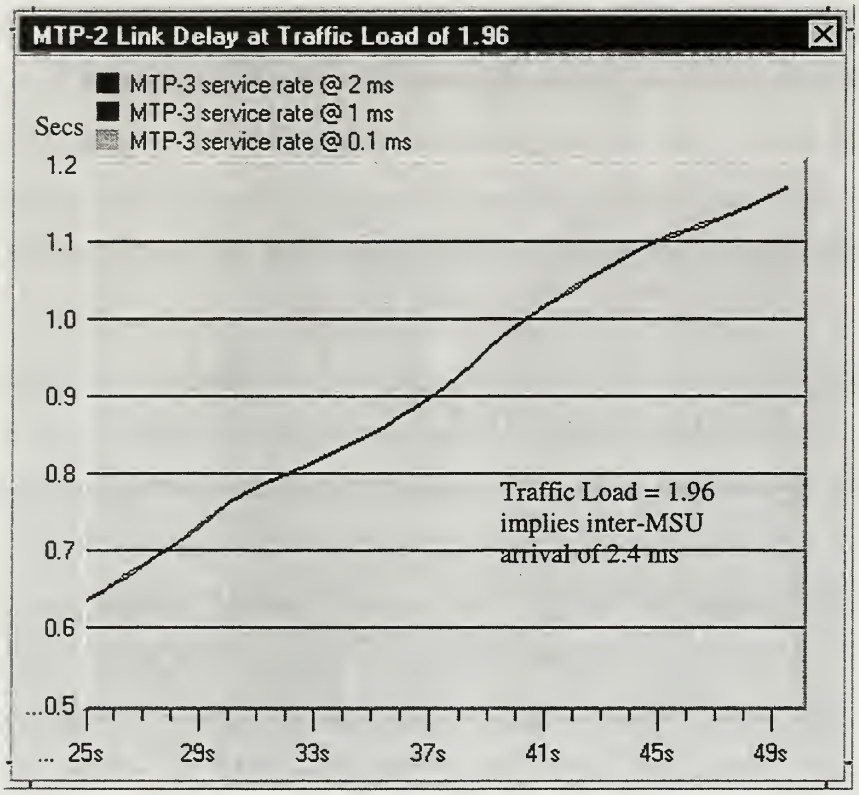

Fig 5.4 MTP-2 Link Congestion at Traffic Load of 1.96

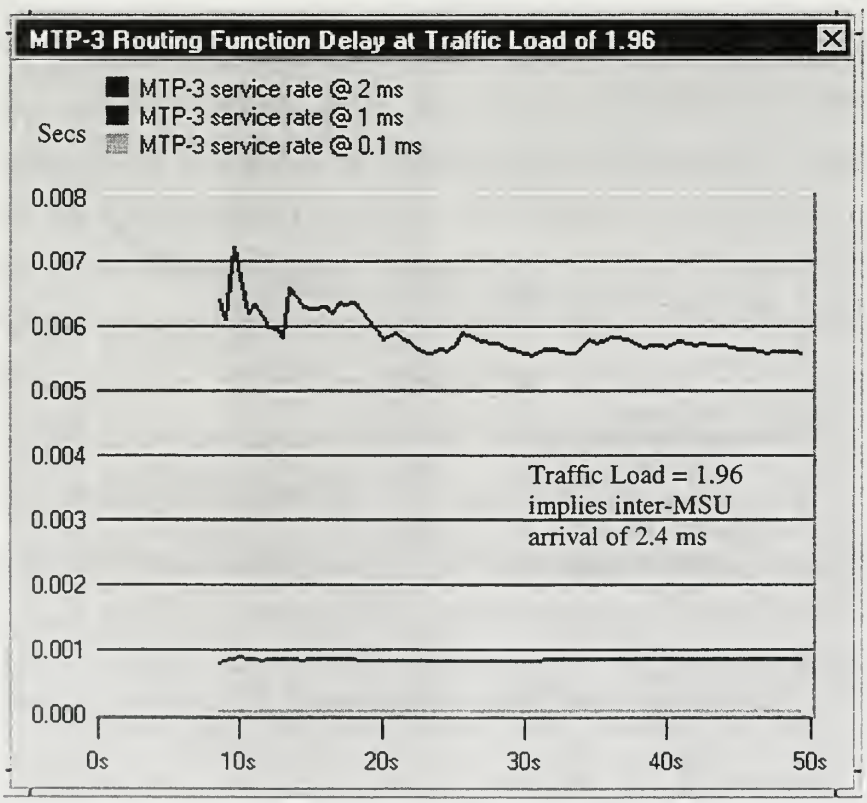

Fig 5.5 MTP-3 Routing Function Delay with MTP-2 congested at Traffic Load of 1.96

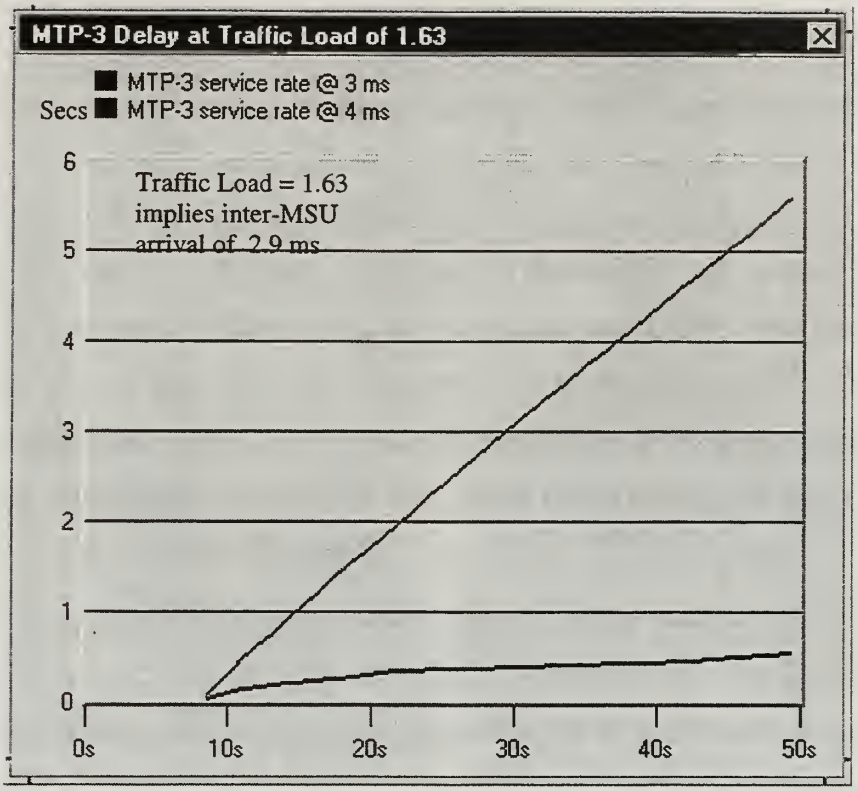

Fig 5.6 MTP-3 Routing Function Congestion at Traffic Load of 1.63

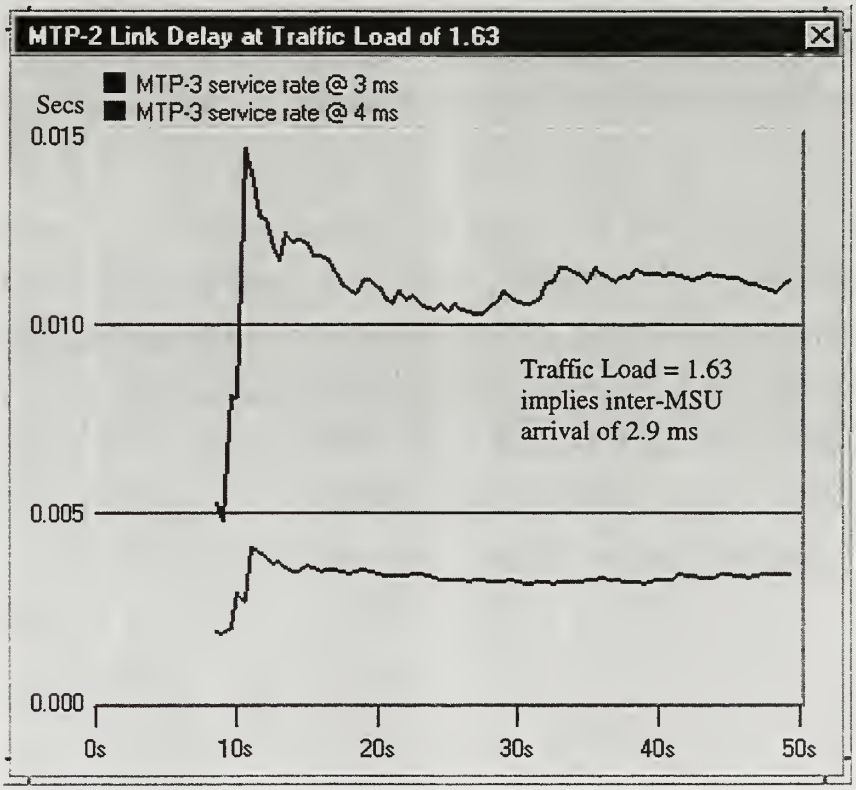

Fig 5.7 MTP-2 Link delay with MTP-3 congested at Traffic Load of 1.63

THIS PAGE IS INTENTIONALLY LEFT BLANK

VI. CONCLUDING REMARKS

In this work, the common channel signaling standard Signaling System No. 7 is studied. The various layers in the protocol, especially the Message Transfer Part (MTP) (i.e., ITU-T Q.703-704) are discussed. The two main protocols in these recommendations are MTP Level 2 and Level 3, which correspond to the datalink and network layer in the OSI model. Modeling of MTP-2 using the simulation tool OPNET is then performed while that of MTP-3 is referenced from Ow's [6] work. The objective is to create simulation models that execute the functions and procedures of the layers as similar to the actual protocol as possible. Together the use of both layers to form SS7 signaling points and signaling networks is also discussed.

Subsequently, the single stage M/G/1 queuing model proposed by Wong [7] for prediction of average Linkset Delay is presented. To illustrate and compare the prediction results against the simulation results, a two-SSP network with 2 signaling links using the OPNET model is constructed and average Linkset Delay is recorded. Results show that the simulation and mathematical prediction are similar when processor service rate for MTP level 3 is fast and not significant to the overall transmission time. Both illustrate an exponential increase in Average Linkset delay when the overall linkset traffic load is increased.

However, the mathematical model that assumes no queuing effects in MTP-3 is not adequate when processor delay in this layer is significant. The simulation shows that this is a critical layer, especially in the routing function. An accumulation of the messages in the buffer of the Routing function can occur at high traffic loads and message units will experience an increasing delay. This shows the simple fact that, in the design of a signaling point especially that of a Signaling Transfer Point, which works as a router, processing in the Routing function has to be carefully studied. Otherwise, it will be a critical bottleneck in the signaling point leading to poor link utilization.

THIS PAGE IS INTENTIONALLY LEFT BLANK

VII. APPENDIX: OPNET SOURCE CODE FOR MTP LEVEL 2

A. HEADER

```
/* ================================================================== */
/* HEADER                                                             */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* ITU-T, Q.703, Jul 1996                                             */
/* The definitions here are for these purpose                         */
/*    (1) General constants                                           */
/*    (2) I/O Stream used for communication with External Modules     */
/*    (3) Queuing buffers allocation and use                          */
/*    (4) Internal Message Codes                                      */
/*    (5) Link and States of the Level 2 Functional Blocks            */
/*    (6) Interrupt Codes and Interrupt Macros                        */
/*    (7) Data Types                                                  */
/*    (8) Function Prototypes                                         */
/* ================================================================== */

/* ================================================================== */
/* (1) General constants                                              */
/* ================================================================== */

/* define TRUE and FALSE */
#define TRUE   1
#define FALSE  0

/* define modular 128 */
#define MOD_SN 128

/* define signaling point types */
#define SP     0
#define STP    1

/* define type of Signaling Units */
#define FISU   0
#define LSSU   1
#define MSU    2

/* define simulation log file limit for link startup */
#define SYS_LINK_STARTUP_LOG_LIMIT 1000

/* ================================================================== */
/* (2) I/O Stream used for communication with External Modules        */
/* ================================================================== */

/* define MTP2 Input/output streams */
#define STREAM_MTP2_LOW_LEVEL_OUT   0
#define STREAM_MTP2_LOW_LEVEL_IN    0
#define STREAM_MTP2_HIGH_LEVEL_OUT  1
#define STREAM_MTP2_HIGH_LEVEL_IN   1

/* ================================================================== */
/* (3) Queuing buffers allocation and use                             */
/*     Note: The buffer numbers used are the OPNET "Queue" module's sub  */
/*           -queue number. The Transfer buffer is for queuing out-   */
/*           going signaling units, while the Retransmission buffer   */
/*           stores every sent unit till an ackowledegment of receipt */
/*           has been recieved. Other buffers are for queuing internal */
```

```
/*          messages for each of the level 2 functional blocks.          */
/* ======================================================================== */

#define TRANSFER_BUFFER       0
#define RETRANSMISSION_BUFFER 1
#define NO_BUFFER             2
#define LSC_BUFFER            3
#define IAC_BUFFER            4
#define POC_BUFFER            5
#define TXC_BUFFER            6
#define RC_BUFFER             7
#define AERM_BUFFER           8
#define SUERM_BUFFER          9
#define CC_BUFFER             10

/* define size of SS7 packet fields */
#define FD_LEN_FLAG    8  /* size of header and trailer Flag   */
#define FD_LEN_BSN     7  /* size of Backward Sequence Number  */
#define FD_LEN_BIB     1  /* size of Backward Indicator Bit    */
#define FD_LEN_FSN     7  /* size of Forward Sequence Number   */
#define FD_LEN_FIB     1  /* size of Forward Indicator Bit     */
#define FD_LEN_LI      8  /* size of Length Indicator (6 used) */
#define FD_LEN_CK      16 /* size of check bits                */
#define FD_LEN_PDU     -1 /* size of PDU set to -1 for encap   */
#define FD_LEN_SF      8  /* size of Status Field              */
#define FD_LEN_SIO     8  /* size of Service Information Octet */
#define FD_LEN_SI      4  /* size of Service Indicator         */
#define FD_LEN_SSF     4  /* size of Sub-Service Field         */
#define FD_LEN_DPC     14 /* size of Destination Point Code    */
#define FD_LEN_OPC     14 /* size of originating Point Code    */
#define FD_LEN_SLS     4  /* size of signaling link selection  */
#define FD_LEN_H0H1    8  /* size of H0H1 field                */
#define FD_LEN_SPARE2  2  /* size of spare field with 2 bits   */
#define FD_LEN_LABEL   32 /* size of a standard label          */
#define FD_LEN_PC      14 /* size of a point code              */

#define FLAG 126

/* define field index for use in OPNET */
#define FD_INDEX_1ST   1
#define FD_INDEX_2ND   2
#define FD_INDEX_3RD   3
#define FD_INDEX_4TH   4
#define FD_INDEX_5TH   5
#define FD_INDEX_6TH   6
#define FD_INDEX_7TH   7
#define FD_INDEX_8TH   8
#define FD_INDEX_9TH   9

/* define the number of overhead bits for a standard Signal Unit (SU) */
/* BSN + BIB + FSN + FIB + LI + CK + 2 flags                          */
/*  7  +  1  +  7  +  1  +  8 + 16 + 16 = 56                          */
#define SU_OVERHEAD 56

/* ======================================================================== */
/* (4) Internal Message Codes                                             */
/* ======================================================================== */

/* definition of SS7 MTP level 2 internal messages for LSC */
#define MSG_IAC_LSC_ALIGNMENT_COMPLETE     1101
#define MSG_IAC_LSC_ALIGNMENT_NOT_POSSIBLE 1102
#define MSG_TXC_LSC_LINK_FAILURE           1103
```

```
#define MSG_RC_LSC_SIO                  1104
#define MSG_RC_LSC_SIN                  1105
#define MSG_RC_LSC_SIE                  1106
#define MSG_RC_LSC_SIOS                 1107
#define MSG_RC_LSC_SIPO                 1108
#define MSG_RC_LSC_FISU_MSU_RECEIVED    1109
#define MSG_RC_LSC_LINK_FAILURE         1110
#define MSG_SUERM_LSC_LINK_FAILURE      1111
#define MSG_POC_LSC_NO_PRO_OUTAGE       1112

/* definition of SS7 MTP level 2 internal messages for IAC */
#define MSG_LSC_IAC_START               1201
#define MSG_LSC_IAC_STOP                1202
#define MSG_LSC_IAC_EMERGENCY           1203
#define MSG_RC_IAC_SIO                  1204
#define MSG_RC_IAC_SIN                  1205
#define MSG_RC_IAC_SIE                  1206
#define MSG_RC_IAC_SIOS                 1207
#define MSG_AERM_IAC_ABORT_PROVING      1208
#define MSG_DAEDR_IAC_CORRECT_SU        1209

/* definition of SS7 MTP level 2 internal messages for POC */
#define MSG_LSC_POC_LOCAL_PRO_OUTAGE    1301
#define MSG_LSC_POC_LOCAL_PRO_RECOVERED 1302
#define MSG_LSC_POC_REMOTE_PRO_OUTAGE   1303
#define MSG_LSC_POC_REMOTE_PRO_RECOVERED 1304
#define MSG_LSC_POC_STOP                1305

/* definition of SS7 MTP level 2 internal messages for TXC */
#define MSG_LSC_TXC_START               1401
#define MSG_LSC_TXC_SEND_SIOS           1402
#define MSG_LSC_TXC_SEND_SIPO           1403
#define MSG_IAC_TXC_SEND_SIO            1404
#define MSG_IAC_TXC_SEND_SIN            1405
#define MSG_IAC_TXC_SEND_SIE            1406
#define MSG_CC_TXC_SEND_SIB             1407
#define MSG_LSC_TXC_SEND_FISU           1408
#define MSG_LSC_TXC_SEND_MSU            1409
#define MSG_RC_TXC_SEND_NACK            1410
#define MSG_RC_TXC_SIB_RECEIVED         1411
#define MSG_RC_TXC_BSNR_AND_BIBR        1412

#define MSG_RC_TXC_FSNX                 1413
#define MSG_LSC_TXC_RETRIEVAL_REQ_FSNC  1414
#define MSG_LSC_TXC_FLUSH_BUFFERS       1415

/* definition of SS7 MTP level 2 internal messages for RC */
#define MSG_LSC_RC_START                1501
#define MSG_LSC_RC_STOP                 1502
#define MSG_LSC_RC_RETRIEVE_BSNT        1503
#define MSG_LSC_RC_RETRIEVE_FSNX        1504
#define MSG_LSC_RC_REJECT_MSU_FISU      1505
#define MSG_LSC_RC_ACCEPT_MSU_FISU      1506
#define MSG_TXC_RC_FSNT                 1507

/* definition of SS7 MTP level 2 internal messages for AERM */
#define MSG_LSC_AERM_SET_Ti_To_Tin      1601
#define MSG_IAC_AERM_SET_Ti_To_Tie      1602
#define MSG_IAC_AERM_START              1603
#define MSG_IAC_AERM_STOP               1604
#define MSG_DAEDR_AERM_SU_ERROR         1605
```

```
/* definition of SS7 MTP level 2 internal messages for SUERM */
#define MSG_LSC_SUERM_START                1701
#define MSG_LSC_SUERM_STOP                 1702
#define MSG_DAEDR_SUERM_SU_ERROR           1703
#define MSG_DAEDR_SUERM_CORRECT_SU         1704

/* definition of SS7 MTP level 2 internal messages for CC */
#define MSG_RC_CC_NORMAL                   1801
#define MSG_RC_CC_BUSY                     1802

/* definition of SS7 MTP messages between Level 2 and 3 functions */
#define MSG_LSAC_LSC_EMERGENCY             8001
#define MSG_LSAC_LSC_EMERGENCY_CEASES      8002
#define MSG_LSAC_LSC_STOP                  8003
#define MSG_LSAC_LSC_START                 8004
#define MSG_LSAC_LSC_FLUSH_BUFFERS         8005
#define MSG_LSAC_LSC_CONTINUE              8006
#define MSG_LSAC_LSC_LOCAL_PRO_OUTAGE      8007
#define MSG_LSAC_LSC_LOCAL_PRO_RECOVERED   8008

#define MSG_TCOC_LSC_RETRIEVE_BSNT         8011
#define MSG_TCOC_LSC_RETRIEVAL_REQ_FSNC    8012

#define MSG_MGMT_LSC_POWER_ON              8021
#define MSG_MGMT_LSC_LEVEL3_FAILURE        8022
#define MSG_MGMT_LSC_LOCAL_PRO_OUTAGE      8023
#define MSG_MGMT_LSC_LOCAL_PRO_RECOVERED   8024

#define MSG_XXX_RC_CONGESTION_DISCARD      8141
#define MSG_XXX_RC_CONGESTION_ACCEPT       8142
#define MSG_XXX_RC_NO_CONGESTION           8143

#define MSG_TXC_TCOC_RETRIEVAL_COMPLETE    8151
#define MSG_RC_TCOC_BSNT                   8152

#define MSG_LSC_LSAC_IN_SERVICE            8161
#define MSG_LSC_LSAC_OUT_SERVICE           8162
#define MSG_LSC_LSAC_REMOTE_PRO_OUTAGE     8163
#define MSG_LSC_LSAC_REMOTE_PRO_RECOVERED  8164

/* ================================================================== */
/* (5) Link and States of the Level 2 Functional Blocks               */
/* ================================================================== */

/* define the link status */
/* states and values assigned are as per SS7 MTP2, Q.703 */
#define LINK_STATUS_OUT_ALIGNMENT      0
#define LINK_STATUS_NORMAL             1
#define LINK_STATUS_EMERGENCY          2
#define LINK_STATUS_OUT_SERVICE        3
#define LINK_STATUS_PROCESSOR_OUTAGE   4
#define LINK_STATUS_BUSY               5

/* define internal states for TXC */
/* states assigned are as per SS7, MTP2, Q.703 */
#define STATE_TXC_IDLE                 1
#define STATE_TXC_IN_SERVICE           2

/* define internal states for LSC */
/* states assigned are as per SS7, MTP2, Q.703 */
#define STATE_LSC_POWER_OFF            1
#define STATE_LSC_ALIGNED_NOT_READY    2
```

```
#define STATE_LSC_ALIGNED_READY        3
#define STATE_LSC_PROCESSOR_OUTAGE     4
#define STATE_LSC_INITIAL_ALIGNMENT    5
#define STATE_LSC_IN_SERVICE           6
#define STATE_LSC_OUT_SERVICE          7

/* define internal states for IAC */
/* states assigned are as per SS7, MTP2, Q.703 */
#define STATE_IAC_IDLE        1
#define STATE_IAC_ALIGNED     2
#define STATE_IAC_NOT_ALIGNED 3
#define STATE_IAC_PROVING     4

/* define internal states for RC */
/* states assigned are as per SS7, MTP2, Q.703 */
#define STATE_RC_IDLE       1
#define STATE_RC_IN_SERVICE 2

/* define internal states for AERM */
/* states assigned are as per SS7, MTP2, Q.703 */
#define STATE_AERM_IDLE       1
#define STATE_AERM_IN_SERVICE 2
#define STATE_AERM_MONITORING 3

/* define internal states for SUERM */
/* states assigned are as per SS7, MTP2, Q.703 */
#define STATE_SUERM_IDLE       1
#define STATE_SUERM_IN_SERVICE 2

/* define internal states for CC */
/* states assigned are as per SS7, MTP2, Q.703 */
#define STATE_CC_IDLE              1
#define STATE_CC_LEVEL2_CONGESTION 2

/* define internal states for POC */
/* states assigned are as per SS7, MTP2, Q.703 */
#define STATE_POC_IDLE              1
#define STATE_POC_LOCAL_PRO_OUTAGE  2
#define STATE_POC_REMOTE_PRO_OUTAGE 3
#define STATE_POC_BOTH_PRO_OUT      4

/* ================================================================== */
/* (6) Interrupt Codes and Interrupt Macros                           */
/* ================================================================== */

/* definition of self interrupt codes */
#define LSC_MSG            1
#define IAC_MSG            2
#define POC_MSG            3
#define TXC_MSG            4
#define RC_MSG             5
#define AERM_MSG           6
#define SUERM_MSG          7
#define CC_MSG             8
#define TXC_TRANS_COMPLETE 9
#define T1_EXPIRE          10
#define T2_EXPIRE          11
#define T3_EXPIRE          12
#define T4_EXPIRE          13
#define T5_EXPIRE          14
#define T6_EXPIRE          15
#define T7_EXPIRE          16
```

```
/* definition of interrupts processed */
#define EXTERNAL_MSG    (op_intrpt_type() == OPC_INTRPT_REMOTE)
#define HIGH_ARRIVAL    ((op_intrpt_type() == OPC_INTRPT_STRM) &&
(op_intrpt_strm() == STREAM_MTP2_HIGH_LEVEL_IN))
#define LOW_ARRIVAL     ((op_intrpt_type() == OPC_INTRPT_STRM) &&
(op_intrpt_strm() == STREAM_MTP2_LOW_LEVEL_IN))
#define TX_MSU          ((op_intrpt_type() == OPC_INTRPT_SELF) &&
(op_intrpt_code() == TXC_TRANS_COMPLETE))
#define LSC             ((op_intrpt_type() == OPC_INTRPT_SELF) &&
((op_intrpt_code() == LSC_MSG) || (op_intrpt_code() == T1_EXPIRE)))
#define IAC             ((op_intrpt_type() == OPC_INTRPT_SELF) &&
((op_intrpt_code() == IAC_MSG) || (op_intrpt_code() == T2_EXPIRE) ||
(op_intrpt_code() == T3_EXPIRE) || (op_intrpt_code() == T4_EXPIRE)))
#define POC             ((op_intrpt_type() == OPC_INTRPT_SELF) &&
(op_intrpt_code() == POC_MSG))
#define TXC             ((op_intrpt_type() == OPC_INTRPT_SELF) &&
((op_intrpt_code() == TXC_MSG) || (op_intrpt_code() == T6_EXPIRE) ||
(op_intrpt_code() == T7_EXPIRE)))
#define RC              ((op_intrpt_type() == OPC_INTRPT_SELF) &&
(op_intrpt_code() == RC_MSG))
#define AERM            ((op_intrpt_type() == OPC_INTRPT_SELF) &&
(op_intrpt_code() == AERM_MSG))
#define SUERM           ((op_intrpt_type() == OPC_INTRPT_SELF) &&
(op_intrpt_code() == SUERM_MSG))
#define CC              ((op_intrpt_type() == OPC_INTRPT_SELF) &&
((op_intrpt_code() == CC_MSG) || (op_intrpt_code() == T5_EXPIRE)))

/* ========================================================================= */
/* (7) Data Types                                                            */
/* ========================================================================= */

/* Definition of a standard timer data structure */
typedef struct {
   Evhandle   ev;      /* The OPNET event handle for the self interrupt */
   int        code;    /* A code value to be returned by the interrupt  */
   double     delay;   /* The time before the timer expires             */
   char       active;  /* True if the timer is running                  */
} timer_type;

/* ========================================================================= */
/* (8) Function Prototypes                                                   */
/* ========================================================================= */

int DAEDR(Packet * pkptr);

int mtp2_increment(int value, int step, int mod_number);
int mtp2_decrement(int value, int step, int mod_number);

int is_BSNR_valid(int FSNF, int FSNT, int BSNR);

void mtp2_timer_start(timer_type * timer);
void mtp2_timer_stop(timer_type * timer);

void mtp2_send_external_msg(Objid target_id, int msg_code, int msg_data, Ici *
iciptr);

void IAC_LSC_ALIGNMENT_COMPLETE();
void IAC_LSC_ALIGNMENT_NOT_POSSIBLE();
void TXC_LSC_LINK_FAILURE();
void RC_LSC_SIO();
void RC_LSC_SIN();
```

```
void RC_LSC_SIE();
void RC_LSC_SIOS();
void RC_LSC_SIPO();
void RC_LSC_FISU_MSU_RECEIVED();
void RC_LSC_LINK_FAILURE();
void SUERM_LSC_LINK_FAILURE();
void POC_LSC_NO_PRO_OUTAGE();

void LSAC_LSC_START();
void LSAC_LSC_STOP();
void LSAC_LSC_EMERGENCY();
void LSAC_LSC_EMERGENCY_CEASES();
void LSAC_LSC_FLUSH_BUFFERS();
void LSAC_LSC_CONTINUE();
void LSAC_LSC_LOCAL_PRO_OUTAGE();
void LSAC_LSC_LOCAL_PRO_RECOVERED();

void TCOC_LSC_RETRIEVE_BSNT();
void TCOC_LSC_RETRIEVAL_REQ_FSNC();

void MGMT_LSC_POWER_ON();
void MGMT_LSC_LEVEL3_FAILURE();
void MGMT_LSC_LOCAL_PRO_OUTAGE();
void MGMT_LSC_LOCAL_PRO_RECOVERED();

void LSC_IAC_START();
void LSC_IAC_STOP();
void LSC_IAC_EMERGENCY();
void RC_IAC_SIO();
void RC_IAC_SIN();
void RC_IAC_SIE();
void RC_IAC_SIOS();
void AERM_IAC_ABORT_PROVING();
void DAEDR_IAC_CORRECT_SU();

void LSC_POC_LOCAL_PRO_OUTAGE();
void LSC_POC_LOCAL_PRO_RECOVERED();
void LSC_POC_REMOTE_PRO_OUTAGE();
void LSC_POC_REMOTE_PRO_RECOVERED();
void LSC_POC_STOP();

void LSC_TXC_START();
void LSC_TXC_SEND_SIOS();
void LSC_TXC_SEND_SIPO();
void IAC_TXC_SEND_SIO();
void IAC_TXC_SEND_SIN();
void IAC_TXC_SEND_SIE();
void CC_TXC_SEND_SIB();
void LSC_TXC_SEND_FISU();
void LSC_TXC_SEND_MSU();
void RC_TXC_SEND_NACK();
void RC_TXC_SIB_RECEIVED();
void RC_TXC_BSNR_AND_BIBR(int bsnr, int bibr);
void RC_TXC_FSNX(int fsnx);
void LSC_TXC_RETRIEVAL_REQ_FSNC();
void LSC_TXC_FLUSH_BUFFERS();

void LSC_RC_START();
void LSC_RC_STOP();
void LSC_RC_RETRIEVE_BSNT();
void LSC_RC_RETRIEVE_FSNX();
void LSC_RC_REJECT_MSU_FISU();
```

```
void LSC_RC_ACCEPT_MSU_FISU();
void TXC_RC_FSNT(int fsnt);

void XXX_RC_CONGESTION_DISCARD();
void XXX_RC_CONGESTION_ACCEPT();
void XXX_RC_NO_CONGESTION();

void LSC_AERM_SET_Ti_To_Tin();
void IAC_AERM_SET_Ti_To_Tie();
void IAC_AERM_START();
void IAC_AERM_STOP();
void DAEDR_AERM_SU_ERROR();

void LSC_SUERM_START();
void LSC_SUERM_STOP();
void DAEDR_SUERM_SU_ERROR();
void DAEDR_SUERM_CORRECT_SU();

void RC_CC_NORMAL();
void RC_CC_BUSY();
```

B. STATE VARIABLES

```
/* Process's own Object ID */
Objid  \own_id;

/* Object ID of expected MTP Level 3 neighbour */
Objid  \mtp3_id;

/* Log file handle for link status */
Log_Handle    \log_hdle_link_status;

/* mtp3's input stream for MSU retrieval */
int    \mtp3_retrieve_input_stream;

/* The servic rate of the link */
double \service_rate;

/* Service time of the LSC Function */
double \LSC_service_time;

/* Service time of the IAC Function */
double \IAC_service_time;

/* Service time of the POC Function */
double \POC_service_time;

/* Service time of the TXC Function */
double \TXC_service_time;

/* Service time of the RC Function */
double \RC_service_time;

/* Service time of the AERM Function */
double \AERM_service_time;

/* Service time of the SUERM Function */
double \SUERM_service_time;

/* Service time of the CC Function */
double \CC_service_time;

/* AERM threshold for normal link status */
int    \Tin;

/* AERM threshold for Emergency Link Status */
int    \Tie;

/* SUERM Threshold */
int    \T;

/* SUERM Parameter */
int    \D;

/* SUERM parameter */
int    \N;

/* No. of Proving Periods allowed by IAC */
int    \M;

/* Alignment Ready Timer for LSC */
double \T1;
```

```
/* Not Aligned Timer for IAC */
double \T2;

/* Aligned Timer for IAC */
double \T3;

/* Normal Proving Period Timer for IAC */
double \T4_Pn;

/* Emergency Period Timer for IAC */
double \T4_Pe;

/* Sending SIB Timer for CC */
double \T5;

/* Remote Congestion TImer for TXC */
double \T6;

/* Excessive Delay in ACK Timer for TXC */
double \T7;

/* ICI pointer for external msg */
Ici *  \iciptr_external;

/* state of LSC */
int    \dv_LSC_state;

/* Local processor outage while not in servive */
int    \dv_LSC_local_processor_outage_during_alignment_flag;

/* Processor Outage while in Service */
int    \dv_LSC_processor_outage_during_service_flag;

/* Emergency link status */
int    \dv_LSC_emergency_flag;

/* Level3 indication received */
int    \dv_LSC_level3_indication_received_flag;

/* Ti value used by LSC */
int    \dv_LSC_Ti;

/* state of IAC */
int    \dv_IAC_state;

/* Emergency status */
int    \dv_IAC_emergency_flag;

/* Further proving required */
int    \dv_IAC_further_proving_flag;

/* Cp counter for IAC */
int    \dv_IAC_Cp;

/* state of POC */
int    \dv_POC_state;

/* state of TXC */
int    \dv_TXC_state;

/* LSSU available for transmission */
```

```
int    \dv_TXC_LSSU_available_flag;

/* RTB is full */
int    \dv_TXC_RTB_full_flag;

/* Link status busy received by RC */
int    \dv_TXC_SIB_received_flag;

/* Inhibit MSU tarnsmission */
int    \dv_TXC_MSU_inhibited_flag;

/* Link status */
int    \dv_TXC_status_indication;

/* FSN of 1st(oldest) MSU in RTB */
int    \dv_TXC_FSNF;

/* FSN of last(newest) MSU in RTB */
int    \dv_TXC_FSNL;

/* FSN of last transmiteed MSU */
int    \dv_TXC_FSNT;

/* FSN of next MSU expected fom remote SP */
int    \dv_TXC_FSNX;

/* FSN of last acceptec MSU by remote SP */
int    \dv_TXC_FSNC;

/* BSN of MSU last recieved by RC */
int    \dv_TXC_BSNR;

/* BSN of last transmitted MSU */
int    \dv_TXC_BSNT;

/* FIB of last transmitted MSU */
int    \dv_TXC_FIBT;

/* BIB of last transmitted MSU */
int    \dv_TXC_BIBT;

/* current FIB value */
int    \dv_TXC_FIB;

/* current BIB value */
int    \dv_TXC_BIB;

/* BIB of MSU last received by RC */
int    \dv_TXC_BIBR;

/* Number of MSUs in TB */
int    \dv_TXC_Cm;

/* state of RC */
int    \dv_RC_state;

/* accept MSU and FISU allowed */
int    \dv_RC_MSU_FISU_accepted_flag;

/* abnormal FIB recieved */
int    \dv_RC_abnormal_FIBR_flag;
```

```
/* abnormal BSN recieved */
int    \dv_RC_abnormal_BSNR_flag;

/* congestion discard mode */
int    \dv_RC_congestion_discard_flag;

/* congestion accept mode */
int    \dv_RC_congestion_accept_flag;

/* Link status received */
int    \dv_RC_status_indication;

/* FSN of 1st MSU in RTB */
int    \dv_RC_FSNF;

/* expected FSN value to receive */
int    \dv_RC_FSNX;

/* expected FIB to receive */
int    \dv_RC_FIBX;

/* transmitted FSN value */
int    \dv_RC_FSNT;

/* transmitted BSN value */
int    \dv_RC_BSNT;

/* retransmission required */
int    \dv_RC_RTR;

/* received FSN value */
int    \dv_RC_FSNR;

/* received FIB value */
int    \dv_RC_FIBR;

/* received BSN value */
int    \dv_RC_BSNR;

/* received BIB value */
int    \dv_RC_BIBR;

/* received Length Indicator value */
int    \dv_RC_LI;

/* recovery from abnormal BIBR for RC */
int    \dv_RC_UNF;

/* recovery from abnormal BSNR for RC */
int    \dv_RC_UNB;

/* state of AERM */
int    \dv_AERM_state;

/* Ti value for AERM */
int    \dv_AERM_Ti;

/* Ca counter for AERM */
int    \dv_AERM_Ca;

/* state of SUERM */
int    \dv_SUERM_state;
```

```
/* Cs counter for SUERM */
int    \dv_SUERM_Cs;

/* Nsu value for SUERM */
int    \dv_SUERM_Nsu;

/* state of CC */
int    \dv_CC_state;

/* timer1 data structure */
timer_type    \dv_timer1;

/* timer2 data structure */
timer_type    \dv_timer2;

/* timer3 data structure */
timer_type    \dv_timer3;

/* timer4 data structure */
timer_type    \dv_timer4;

/* timer5 data structure */
timer_type    \dv_timer5;

/* timer6 data structure */
timer_type    \dv_timer6;

/* timer7 data structure */
timer_type    \dv_timer7;

/* SU transmission request flag */
int    \trans_req;

/* Pointer for packet to be transmitted */
Packet *      \pkptr_tx;

/* time link is in service */
double \start_time;

/* total bits received */
double \total_bits_rcv;

/* statistic handle for link utilization */
Stathandle    \link_utilization_handle;
```

C. TEMPORARY VARIABLES

```
Packet * pkptr;
Packet * pkptr_tx_pdu;
Packet * pkptr_rx;
Packet * pkptr_rx_pdu;
int      packet_no_error;
int      msg_code;
int      SU_type_to_send;
double   SU_len;
double   SU_tx_time;
Ici *    iciptr_input;
char str1[10];
int i;
```

D. FUNCTION DECLARATION

```
/* ================================================================== */
/* FUNCTIONS                                                          */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* ITU-T, Q.703, Jul 1996                                             */
/* The functions here includes                                        */
/*   (1) DAEDR transmission Error Function                            */
/*   (2) Binary increment and decrement functions                     */
/*   (3) Timer start/stop functions                                   */
/*   (4) Inter-level Communication Function                           */
/*   (5) Packet Queuing functions for all internal messages           */
/* ================================================================== */

/* ================================================================== */
/* (1) DAEDR transmission Error Function                              */
/* ================================================================== */

/* This function represents the Delimitation, Alignment and Error */
/* Detection (receiving), DAEDR function of MTP2. It is executed  */
/* everytime a SU is received from the link, it should decide      */
/* whether the SU packet is correct or in error. Channel error     */
/* characteristics can be simulated by this function. Should the   */
/* function decides that the SU is correct it sends                */
/* "DAEDR_IAC_CORRECT_SU" and "DAEDR_SUERM_CORRECT_SU" else it     */
/* "DAEDR_SUERM_SU_ERROR" and "DAEDR_AERM_SU_ERROR". In this       */
/* implementation the former is used assuming zero channel error   */
/* but should be modified as per requirement otherwise             */

int DAEDR(pkptr)
   Packet * pkptr;
{
   int status;

   FIN(DAEDR(pkptr));

   DAEDR_IAC_CORRECT_SU();
   DAEDR_SUERM_CORRECT_SU();
   status = TRUE;

   FRET(status);
}

/* ================================================================== */
/* (2) Binary increment and decrement functions                       */
/* ================================================================== */

/* define Binary increment */
int mtp2_increment(value, step, mod_number)
   int value;
   int step;
   int mod_number;
{
   int result;
   FIN (mtp2_increment(value, step, mod_number))

   result = (value + step) % mod_number;

   FRET (result);
}
```

```
/* define Binary decrement */
int mtp2_decrement(value, step, mod_number)
   int value;
   int step;
   int mod_number;
{
   int result;

   FIN (mtp2_decrement(value, step, mod_number));

   if ((value - step) < 0) {
      result = mod_number - (step - value);
   }
   else {
     result = value - step;
   }

   FRET (result);
}

/* this function checks whether BSNR is valid or not */
/* it has to be between FSNF-1 and FSNT Mod          */
/* it is used by RC for MSU checks                   */

int is_BSNR_valid(FSNF, FSNT, BSNR)
   int FSNF;
   int FSNT;
   int BSNR;
{
   int status;
   int end;
   int lowerlimit;

   FIN(is_BSNR_valid(FSNF, FSNT, BSNR));

   status = FALSE;
   end = FALSE;
   lowerlimit = mtp2_decrement(FSNF, 1, MOD_SN);
   while (!status && !end) {
      if (BSNR == lowerlimit) {
         status = TRUE;
      }
      if (lowerlimit == FSNT) {
         end = TRUE;
      }
      lowerlimit = mtp2_increment(lowerlimit, 1, MOD_SN);
   }

   FRET(status);
}


/* ================================================================== */
/* (3) Timer start/stop functions                                     */
/* ================================================================== */

/* These functions emulate a "timer" by using OPNET's self interrupts */

void mtp2_timer_start(timer)
   timer_type * timer;
{
```

```
    /* Start a timer, by queueing a self interrupt request.       */
    /* stop it first and restart if the timer is already running; */
    FIN (mtp2_timer_start (timer));

    if (timer->active) {
       op_ev_cancel(timer->ev);
    }
    timer->ev = op_intrpt_schedule_self(op_sim_time() + timer->delay,
       timer->code);
    timer->active = 1;

    FOUT;
}

void mtp2_timer_stop(timer)
    timer_type * timer;
{
    /* Stop a timer by canceling its outstanding interrupt request.  */
    /* (Don't cancel the interrupt if it is the current one, as this */
    /* causes an error in the simulation kernel.)                    */
    FIN (mtp2_timer_stop(timer));

    if (timer->active && !(op_intrpt_type() == OPC_INTRPT_SELF &&
       op_intrpt_code() == timer->code)) {
       op_ev_cancel(timer->ev);
    }
    timer->active = 0;

    FOUT;
}

/* ================================================================== */
/* (4) Inter-level Communication Function                             */
/* ================================================================== */

/* send an external message function */
void mtp2_send_external_msg(target_id, msg_code, msg_data, iciptr)
    Objid target_id;
    int msg_code;
    int msg_data;
    Ici * iciptr;
{
    FIN (mtp2_send_external_msg(target_id, msg_code, msg_data, iciptr));

    /* set the ICI data and install it to the current interrupt */
    op_ici_attr_set(iciptr, "data", msg_data);
    op_ici_install(iciptr);

    /* activate the interrupt to send out the message code and data value */
    op_intrpt_force_remote(msg_code, target_id);

    /* de-install the ICI so the they do not appear in future interrupts */
    op_ici_install(OPC_NIL);

   FOUT;
}
```

```
/* print a warning message */
void ss7_mtp2_warn (msg)
   char *     msg;
{

   /* Print a warning message */
   FIN (ss7_mtp2_warn(msg));

   op_prg_odb_print_major ("Warning from SS7 mtp2 functions:", msg, OPC_NIL);

   FOUT;
}

/* ========================================================================= */
/* (5) Packet Queuing functions for all internal messages                    */
/* ========================================================================= */

/* define packet queuing functions for messages send to LSC */

void IAC_LSC_ALIGNMENT_COMPLETE()
{
   Packet * pkptr;

   FIN(IAC_LSC_ALIGNMENT_COMPLETE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_IAC_LSC_ALIGNMENT_COMPLETE, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void IAC_LSC_ALIGNMENT_NOT_POSSIBLE()
{
   Packet * pkptr;

   FIN(IAC_LSC_ALIGNMENT_NOT_POSSIBLE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_IAC_LSC_ALIGNMENT_NOT_POSSIBLE, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void TXC_LSC_LINK_FAILURE()
{
   Packet * pkptr;

   FIN(TXC_LSC_LINK_FAILURE());

   pkptr = op_pk_create(0);
```

```
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_DOUBLE,
MSG_TXC_LSC_LINK_FAILURE, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void RC_LSC_SIO()
{
   Packet * pkptr;

   FIN(RC_LSC_SIO());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_LSC_SIO,
16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void RC_LSC_SIN()
{
   Packet * pkptr;

   FIN(RC_LSC_SIN());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_LSC_SIN,
16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void RC_LSC_SIE()
{
   Packet * pkptr;

   FIN(RC_LSC_SIE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_LSC_SIE,
16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void RC_LSC_SIOS()
{
   Packet * pkptr;

   FIN(RC_LSC_SIOS());
```

```
   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_LSC_SIOS,
16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void RC_LSC_SIPO()
{
   Packet * pkptr;

   FIN(RC_LSC_SIPO());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_LSC_SIPO,
16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void RC_LSC_FISU_MSU_RECEIVED()
{
   Packet * pkptr;

   FIN(RC_LSC_FISU_MSU_RECEIVED());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_RC_LSC_FISU_MSU_RECEIVED, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void RC_LSC_LINK_FAILURE()
{
   Packet * pkptr;

   FIN(RC_LSC_LINK_FAILURE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_RC_LSC_LINK_FAILURE, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void SUERM_LSC_LINK_FAILURE()
{
   Packet * pkptr;
```

```
   FIN(SUERM_LSC_LINK_FAILURE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_SUERM_LSC_LINK_FAILURE, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void POC_LSC_NO_PRO_OUTAGE()
{
   Packet * pkptr;

   FIN(POC_LSC_NO_PRO_OUTAGE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_POC_LSC_NO_PRO_OUTAGE, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSAC_LSC_START()
{
   Packet * pkptr;

   FIN(LSAC_LSC_START());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSAC_LSC_START, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSAC_LSC_STOP()
{
   Packet * pkptr;

   FIN(LSAC_LSC_STOP());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_LSAC_LSC_STOP,
16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSAC_LSC_EMERGENCY()
```

```
{
   Packet * pkptr;

   FIN(LSAC_LSC_EMERGENCY());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSAC_LSC_EMERGENCY, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSAC_LSC_EMERGENCY_CEASES()
{
   Packet * pkptr;

   FIN(LSAC_LSC_EMERGENCY_CEASES());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSAC_LSC_EMERGENCY_CEASES, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSAC_LSC_FLUSH_BUFFERS()
{
   Packet * pkptr;

   FIN(LSAC_LSC_FLUSH_BUFFERS());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSAC_LSC_FLUSH_BUFFERS, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSAC_LSC_CONTINUE()
{
   Packet * pkptr;

   FIN(LSAC_LSC_CONTINUE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSAC_LSC_CONTINUE, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   )

   FOUT;
}
```

```
void LSAC_LSC_LOCAL_PRO_OUTAGE()
{
   Packet * pkptr;

   FIN(LSAC_LSC_LOCAL_PRO_OUTAGE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSAC_LSC_LOCAL_PRO_OUTAGE, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSAC_LSC_LOCAL_PRO_RECOVERED()
{
   Packet * pkptr;

   FIN(LSAC_LSC_LOCAL_PRO_RECOVERED());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSAC_LSC_LOCAL_PRO_RECOVERED, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void TCOC_LSC_RETRIEVE_BSNT()
{
   Packet * pkptr;

   FIN(TCOC_LSC_RETRIEVE_BSNT());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_TCOC_LSC_RETRIEVE_BSNT, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void TCOC_LSC_RETRIEVAL_REQ_FSNC()
{
   Packet * pkptr;

   FIN(TCOC_LSC_RETRIEVAL_REQ_FSNC());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_TCOC_LSC_RETRIEVAL_REQ_FSNC, 16);
   if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }
```

```
    FOUT;
}

void MGMT_LSC_POWER_ON()
{
    Packet * pkptr;

    FIN(MGMT_LSC_POWER_ON());

    pkptr = op_pk_create(0);
    op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_MGMT_LSC_POWER_ON, 16);
    if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
    }

    FOUT;
}

void MGMT_LSC_LEVEL3_FAILURE()
{
    Packet * pkptr;

    FIN(MGMT_LSC_LEVEL3_FAILURE());

    pkptr = op_pk_create(0);
    op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_MGMT_LSC_LEVEL3_FAILURE, 16);
    if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
    }

    FOUT;
}

void MGMT_LSC_LOCAL_PRO_OUTAGE()
{
    Packet * pkptr;

    FIN(MGMT_LSC_LOCAL_PRO_OUTAGE());

    pkptr = op_pk_create(0);
    op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_MGMT_LSC_LOCAL_PRO_OUTAGE, 16);
    if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
    }

    FOUT;
}

void MGMT_LSC_LOCAL_PRO_RECOVERED()
{
    Packet * pkptr;

    FIN(MGMT_LSC_LOCAL_PRO_RECOVERED());

    pkptr = op_pk_create(0);
    op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_MGMT_LSC_LOCAL_PRO_RECOVERED, 16);
    if (op_subq_pk_insert(LSC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
```

```
    }

    FOUT;
}

/* define packet queuing functions for messages send to IAC */
void LSC_IAC_START()
{
    Packet * pkptr;

    FIN(LSC_IAC_START());

    pkptr = op_pk_create(0);
    op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_LSC_IAC_START,
16);
    if (op_subq_pk_insert(IAC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
    }

    FOUT;
}

void LSC_IAC_STOP()
{
    Packet * pkptr;

    FIN(LSC_IAC_STOP());

    pkptr = op_pk_create(0);
    op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_LSC_IAC_STOP,
16);
    if (op_subq_pk_insert(IAC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
    }

    FOUT;
}

void LSC_IAC_EMERGENCY()
{
    Packet * pkptr;

    FIN(LSC_IAC_EMERGENCY());

    pkptr = op_pk_create(0);
    op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_IAC_EMERGENCY, 16);
    if (op_subq_pk_insert(IAC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
    }

    FOUT;
}

void RC_IAC_SIO()
{
    Packet * pkptr;

    FIN(RC_IAC_SIO());

    pkptr = op_pk_create(0);
```

```
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_IAC_SIO,
16);
   if (op_subq_pk_insert(IAC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void RC_IAC_SIN()
{
   Packet * pkptr;

   FIN(RC_IAC_SIN());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_IAC_SIN,
16);
   if (op_subq_pk_insert(IAC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void RC_IAC_SIE()
{
   Packet * pkptr;

   FIN(RC_IAC_SIE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_IAC_SIE,
16);
   if (op_subq_pk_insert(IAC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void RC_IAC_SIOS()
{
   Packet * pkptr;

   FIN(RC_IAC_SIOS());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_IAC_SIOS,
16);
   if (op_subq_pk_insert(IAC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void AERM_IAC_ABORT_PROVING()
{
   Packet * pkptr;

   FIN(AERM_IAC_ABORT_PROVING());
```

```
   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_AERM_IAC_ABORT_PROVING, 16);
   if (op_subq_pk_insert(IAC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}



void DAEDR_IAC_CORRECT_SU()
{
   Packet * pkptr;

   FIN(DAEDR_IAC_CORRECT_SU());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_DAEDR_IAC_CORRECT_SU, 16);
   if (op_subq_pk_insert(IAC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

/* define packet queuing functions for messages send to POC */
void LSC_POC_LOCAL_PRO_OUTAGE()
{
   Packet * pkptr;

   FIN(LSC_POC_LOCAL_PRO_OUTAGE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_POC_LOCAL_PRO_OUTAGE, 16);
   if (op_subq_pk_insert(POC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_POC_LOCAL_PRO_RECOVERED()
{
   Packet * pkptr;

   FIN(LSC_POC_LOCAL_PRO_RECOVERED());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_POC_LOCAL_PRO_RECOVERED, 16);
   if (op_subq_pk_insert(POC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}
```

```
void LSC_POC_REMOTE_PRO_OUTAGE()
{
   Packet * pkptr;

   FIN(LSC_POC_REMOTE_PRO_OUTAGE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_POC_REMOTE_PRO_OUTAGE, 16);
   if (op_subq_pk_insert(POC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }
   FOUT;
}

void LSC_POC_REMOTE_PRO_RECOVERED()
{
   Packet * pkptr;

   FIN(LSC_POC_REMOTE_PRO_RECOVERED());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_POC_REMOTE_PRO_RECOVERED, 16);
   if (op_subq_pk_insert(POC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_POC_STOP()
{
   Packet * pkptr;

   FIN(LSC_POC_STOP());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_LSC_POC_STOP,
16);
   if (op_subq_pk_insert(POC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

/* define packet queuing functions for messages send to TXC */
void LSC_TXC_START()
{
   Packet * pkptr;

   FIN(LSC_TXC_START());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_LSC_TXC_START,
16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
```

```
}

void LSC_TXC_SEND_SIOS()
{
   Packet * pkptr;

   FIN(LSC_TXC_SEND_SIOS());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_TXC_SEND_SIOS, 16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_TXC_SEND_SIPO()
{
   Packet * pkptr;

   FIN(LSC_TXC_SEND_SIPO());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_TXC_SEND_SIPO, 16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void IAC_TXC_SEND_SIO()
{
   Packet * pkptr;

   FIN(IAC_TXC_SEND_SIO());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_IAC_TXC_SEND_SIO, 16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void IAC_TXC_SEND_SIN()
{
   Packet * pkptr;

   FIN(IAC_TXC_SEND_SIN());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_IAC_TXC_SEND_SIN, 16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }
```

```
   FOUT;
}

void IAC_TXC_SEND_SIE()
{
   Packet * pkptr;

   FIN(IAC_TXC_SEND_SIE());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_IAC_TXC_SEND_SIE, 16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}


void CC_TXC_SEND_SIB()
{
   Packet * pkptr;

   FIN(CC_TXC_SEND_SIB());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_CC_TXC_SEND_SIB, 16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_TXC_SEND_FISU()
{
   Packet * pkptr;

   FIN(LSC_TXC_SEND_FISU());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_TXC_SEND_FISU, 16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_TXC_SEND_MSU()
{
   Packet * pkptr;

   FIN(LSC_TXC_SEND_MSU());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_TXC_SEND_MSU, 16);
```

```
    if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
    }

    FOUT;
}

void RC_TXC_SEND_NACK()
{
    Packet * pkptr;

    FIN(RC_TXC_SEND_NACK());

    pkptr = op_pk_create(0);
    op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_RC_TXC_SEND_NACK, 16);
    if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
    }

    FOUT;
}

void RC_TXC_SIB_RECEIVED()
{
    Packet * pkptr;

    FIN(RC_TXC_SIB_RECEIVED());

    pkptr = op_pk_create(0);
    op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_RC_TXC_SIB_RECEIVED, 16);
    if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
    }

    FOUT;
}

void RC_TXC_BSNR_AND_BIBR(bsnr, bibr)
    int bsnr;
    int bibr;
{
    Packet * pkptr;

    FIN(RC_TXC_BSNR_AND_BIBR(BSNR, BIBR));

    pkptr = op_pk_create(0);
    op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_RC_TXC_BSNR_AND_BIBR, 16);
    op_pk_fd_set(pkptr, FD_INDEX_2ND, OPC_FIELD_TYPE_INTEGER, bsnr, 16);
    op_pk_fd_set(pkptr, FD_INDEX_3RD, OPC_FIELD_TYPE_INTEGER, bibr, 16);
    if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
       op_pk_destroy(pkptr);
    }

    FOUT;
}

void RC_TXC_FSNX(fsnx)
    int fsnx;
{
```

```
   Packet * pkptr;

   FIN(RC_TXC_FSNX(fsnx));

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_TXC_FSNX,
16);
   op_pk_fd_set(pkptr, FD_INDEX_2ND, OPC_FIELD_TYPE_INTEGER, fsnx, 16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_TXC_RETRIEVAL_REQ_FSNC()
{
   Packet * pkptr;

   FIN(LSC_TXC_RETRIEVAL_REQ_FSNC());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_TXC_RETRIEVAL_REQ_FSNC, 16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_TXC_FLUSH_BUFFERS()
{
   Packet * pkptr;

   FIN(LSC_TXC_FLUSH_BUFFERS());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_TXC_FLUSH_BUFFERS, 16);
   if (op_subq_pk_insert(TXC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

/* define packet queuing functions for messages send to RC */
void LSC_RC_START()
{
   Packet * pkptr;

   FIN(LSC_RC_START());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_LSC_RC_START,
16);
   if (op_subq_pk_insert(RC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
```

```
}

void LSC_RC_STOP()
{
   Packet * pkptr;

   FIN(LSC_RC_STOP());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_LSC_RC_STOP,
16);
   if (op_subq_pk_insert(RC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_RC_RETRIEVE_BSNT()
{
   Packet * pkptr;

   FIN(LSC_RC_RETRIEVE_BSNT());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_RC_RETRIEVE_BSNT, 16);
   if (op_subq_pk_insert(RC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_RC_RETRIEVE_FSNX()
{
   Packet * pkptr;

   FIN(LSC_RC_RETRIEVE_FSNX());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_RC_RETRIEVE_FSNX, 16);
   if (op_subq_pk_insert(RC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_RC_REJECT_MSU_FISU()
{
   Packet * pkptr;

   FIN(LSC_RC_REJECT_MSU_FISU());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_RC_REJECT_MSU_FISU, 16);
   if (op_subq_pk_insert(RC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }
```

```
   FOUT;
}

void LSC_RC_ACCEPT_MSU_FISU()
{
   Packet * pkptr;

   FIN(LSC_RC_ACCEPT_MSU_FISU());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_RC_ACCEPT_MSU_FISU, 16);
   if (op_subq_pk_insert(RC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void TXC_RC_FSNT(fsnt)
   int fsnt;
{
   Packet * pkptr;

   FIN(TXC_RC_FSNT());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_TXC_RC_FSNT,
16);
   op_pk_fd_set(pkptr, FD_INDEX_2ND, OPC_FIELD_TYPE_INTEGER, fsnt, 16);
   if (op_subq_pk_insert(RC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}


void XXX_RC_CONGESTION_DISCARD()
{
   Packet * pkptr;

   FIN(XXX_RC_CONGESTION_DISCARD());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_XXX_RC_CONGESTION_DISCARD, 16);
   if (op_subq_pk_insert(RC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void XXX_RC_CONGESTION_ACCEPT()
{
   Packet * pkptr;

   FIN(XXX_RC_CONGESTION_ACCEPT());

   pkptr = op_pk_create(0);
```

```
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_XXX_RC_CONGESTION_ACCEPT, 16);
   if (op_subq_pk_insert(RC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void XXX_RC_NO_CONGESTION()
{
   Packet * pkptr;

   FIN(XXX_RC_NO_CONGESTION());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_XXX_RC_NO_CONGESTION, 16);
   if (op_subq_pk_insert(RC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}


/* define packet queuing functions for messages send to AERM */
void IAC_AERM_SET_Ti_To_Tie()
{
   Packet * pkptr;

   FIN(IAC_AERM_SET_Ti_To_Tie());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_IAC_AERM_SET_Ti_To_Tie, 16);
   if (op_subq_pk_insert(AERM_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void LSC_AERM_SET_Ti_To_Tin()
{
   Packet * pkptr;

   FIN(LSC_AERM_SET_Ti_To_Tin());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_AERM_SET_Ti_To_Tin, 16);
   if (op_subq_pk_insert(AERM_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void IAC_AERM_START()
{
   Packet * pkptr;
```

```
   FIN(IAC_AERM_START());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_IAC_AERM_START, 16);
   if (op_subq_pk_insert(AERM_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void IAC_AERM_STOP()
{
   Packet * pkptr;

   FIN(IAC_AERM_STOP());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_IAC_AERM_STOP,
16);
   if (op_subq_pk_insert(AERM_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void DAEDR_AERM_SU_ERROR()
{
   Packet * pkptr;

   FIN(DAEDR_AERM_SU_ERROR());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_DAEDR_AERM_SU_ERROR, 16);
   if (op_subq_pk_insert(AERM_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

/* define packet queuing functions for messages send to SUERM */
void LSC_SUERM_START()
{
   Packet * pkptr;

   FIN(LSC_SUERM_START());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_SUERM_START, 16);
   if (op_subq_pk_insert(SUERM_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}
```

```
void LSC_SUERM_STOP()
{
   Packet * pkptr;

   FIN(LSC_SUERM_STOP());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_LSC_SUERM_STOP, 16);
   if (op_subq_pk_insert(SUERM_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void DAEDR_SUERM_SU_ERROR()
{
   Packet * pkptr;

   FIN(DAEDR_SUERM_SU_ERROR());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_DAEDR_SUERM_SU_ERROR, 16);
   if (op_subq_pk_insert(SUERM_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

void DAEDR_SUERM_CORRECT_SU()
{
   Packet * pkptr;

   FIN(DAEDR_SUERM_CORRECT_SU());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER,
MSG_DAEDR_SUERM_CORRECT_SU, 16);
   if (op_subq_pk_insert(SUERM_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}

/* define packet queuing functions for messages send to CC */
void RC_CC_NORMAL()
{
   Packet * pkptr;

   FIN(RC_CC_NORMAL());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_CC_NORMAL,
16);
   if (op_subq_pk_insert(CC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }
```

```
   FOUT;
}

void RC_CC_BUSY()
{
   Packet * pkptr;

   FIN(RC_CC_BUSY());

   pkptr = op_pk_create(0);
   op_pk_fd_set(pkptr, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, MSG_RC_CC_BUSY,
16);
   if (op_subq_pk_insert(CC_BUFFER, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK) {
      op_pk_destroy(pkptr);
   }

   FOUT;
}
```

E. EXECUTIVE FOR "init" STATE

```
/* ========================================================================== */
/* Process Init State                                                         */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)             */
/* ITU-T, Q.703, Jul 1996                                                     */
/* ========================================================================== */
/* ========================================================================== */
/* (1) Retrieving of Process parameters                                       */
/* ========================================================================== */

/* get queue module's own object id and that of MTP3 */
own_id = op_id_self();
mtp3_id = op_id_from_name(op_topo_parent(own_id), OPC_OBJTYPE_QUEUE, "MTP3");

/* get assigned values for all the model attribute */
op_ima_obj_attr_get(own_id, "service_rate", &service_rate);
op_ima_obj_attr_get(own_id, "Tin", &Tin);
op_ima_obj_attr_get(own_id, "Tie", &Tie);
op_ima_obj_attr_get(own_id, "T", &T);
op_ima_obj_attr_get(own_id, "D", &D);
op_ima_obj_attr_get(own_id, "N", &N);
op_ima_obj_attr_get(own_id, "M", &M);
op_ima_obj_attr_get(own_id, "Timer1", &T1);
op_ima_obj_attr_get(own_id, "Timer2", &T2);
op_ima_obj_attr_get(own_id, "Timer3", &T3);
op_ima_obj_attr_get(own_id, "Timer4_Pn", &T4_Pn);
op_ima_obj_attr_get(own_id, "Timer4_Pe", &T4_Pe);
op_ima_obj_attr_get(own_id, "Timer5", &T5);
op_ima_obj_attr_get(own_id, "Timer6", &T6);
op_ima_obj_attr_get(own_id, "Timer7", &T7);
op_ima_obj_attr_get(own_id, "LSC_service_time", &LSC_service_time);
op_ima_obj_attr_get(own_id, "IAC_service_time", &IAC_service_time);
op_ima_obj_attr_get(own_id, "POC_service_time", &POC_service_time);
op_ima_obj_attr_get(own_id, "TXC_service_time", &TXC_service_time);
op_ima_obj_attr_get(own_id, "RC_service_time", &RC_service_time);
op_ima_obj_attr_get(own_id, "AERM_service_time", &AERM_service_time);
op_ima_obj_attr_get(own_id, "SUERM_service_time", &SUERM_service_time);
op_ima_obj_attr_get(own_id, "CC_service_time", &CC_service_time);

/* ========================================================================== */
/* (2) Initiailization of Process variables                                   */
/* ========================================================================== */

/* convert service time from milliseconds to seconds */
LSC_service_time = LSC_service_time / 1000;
IAC_service_time = IAC_service_time / 1000;
POC_service_time = POC_service_time / 1000;
TXC_service_time = TXC_service_time / 1000;
RC_service_time  = RC_service_time  / 1000;
AERM_service_time = AERM_service_time  / 1000;
SUERM_service_time = SUERM_service_time / 1000;
CC_service_time = CC_service_time / 1000;

/* define data class for the simulation log */
_itoa(own_id, str1, 10);

log_hdle_link_status = op_prg_log_handle_create(
                          OpC_Log_Category_Protocol,
                          "MTP2 - Link Status",
                          str1,
```

```
                                SYS_LINK_STARTUP_LOG_LIMIT);

/* define statistic handle for link utilization */
link_utilization_handle = op_stat_reg("link_utilization", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);

/* init Ici's */
iciptr_external = op_ici_create("ss7_ici_fmt");

/* init timer's */
dv_timer1.code = T1_EXPIRE;
dv_timer1.delay = T1;

dv_timer2.code = T2_EXPIRE;
dv_timer2.delay = T2;

dv_timer3.code = T3_EXPIRE;
dv_timer3.delay = T3;

dv_timer4.code = T4_EXPIRE;
dv_timer4.delay = T4_Pn;

dv_timer5.code = T5_EXPIRE;
dv_timer5.delay = T5;

dv_timer6.code = T6_EXPIRE;
dv_timer6.delay = T6;

dv_timer7.code = T7_EXPIRE;
dv_timer7.delay = T7;

/* initialize the states of all the functional blocks */
dv_LSC_state   = STATE_LSC_POWER_OFF;
dv_IAC_state   = STATE_IAC_IDLE;
dv_POC_state   = STATE_POC_IDLE;
dv_TXC_state   = STATE_TXC_IDLE;
dv_RC_state    = STATE_RC_IDLE;
dv_AERM_state  = STATE_AERM_IDLE;
dv_SUERM_state = STATE_SUERM_IDLE;
dv_CC_state    = STATE_CC_IDLE;

/* ================================================================== */
/* (3) Power up link and processors and wait for START command from MTP3 */
/* ================================================================== */

/* start message processing for all functional blocks */
op_intrpt_schedule_self(op_sim_time() + LSC_service_time, LSC_MSG);
op_intrpt_schedule_self(op_sim_time() + IAC_service_time, IAC_MSG);
op_intrpt_schedule_self(op_sim_time() + POC_service_time, POC_MSG);
op_intrpt_schedule_self(op_sim_time() + TXC_service_time, TXC_MSG);
op_intrpt_schedule_self(op_sim_time() + RC_service_time, RC_MSG);
op_intrpt_schedule_self(op_sim_time() + AERM_service_time, AERM_MSG);
op_intrpt_schedule_self(op_sim_time() + SUERM_service_time, SUERM_MSG);
op_intrpt_schedule_self(op_sim_time() + CC_service_time, CC_MSG);

/* start Signaling Unit transmission routine */
op_intrpt_schedule_self (op_sim_time() + 0.001, TXC_TRANS_COMPLETE);

/* Enable transmission of Signaling Units */
trans_req = FALSE;

/* Power up LSC, this is needed unless an external MGMT function */
```

```
/* is implemented to start the link                                    */
MGMT_LSC_POWER_ON();
```

F. EXECUTIVE FOR "ext_msg" STATE

```
/* ================================================================== */
/* External Message Processing State                                  */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* ITU-T, Q.703, Jul 1996                                             */
/* Note:                                                              */
/*    This state receives all incoming messages for MTP2.             */
/*    Messages here do not include SS7 Signaling Units.               */
/*    All Messages intended for this level are received here and      */
/*    being recognized as an interrupt with type "OPC_INTRPT_REMOTE". */
/*    The "code" field of the interrupt are then retrieved to determine */
/*    the message and if necessary, the "ici" field of the interrupt  */
/*    can also be read (if more data are also sent).                  */
/*    All received messages are then queued into message buffer of the */
/*    respective functional block for processing.                     */
/* ================================================================== */

iciptr_input = op_intrpt_ici();
switch (op_intrpt_code()) {

   case MSG_LSAC_LSC_START:
      LSAC_LSC_START();
      break;

   case MSG_LSAC_LSC_STOP:
      LSAC_LSC_STOP();
      break;

   case MSG_LSAC_LSC_EMERGENCY:
      LSAC_LSC_EMERGENCY();
      break;

   case MSG_LSAC_LSC_EMERGENCY_CEASES:
      LSAC_LSC_EMERGENCY_CEASES();
      break;

   case MSG_LSAC_LSC_LOCAL_PRO_OUTAGE:
      LSAC_LSC_LOCAL_PRO_OUTAGE();
      break;

   case MSG_LSAC_LSC_LOCAL_PRO_RECOVERED:
      LSAC_LSC_LOCAL_PRO_RECOVERED();
      break;

   case MSG_LSAC_LSC_FLUSH_BUFFERS:
      LSAC_LSC_FLUSH_BUFFERS();
      break;

   case MSG_LSAC_LSC_CONTINUE:
      LSAC_LSC_CONTINUE();
      break;

   case MSG_TCOC_LSC_RETRIEVE_BSNT:
      TCOC_LSC_RETRIEVE_BSNT();
      break;

   case MSG_TCOC_LSC_RETRIEVAL_REQ_FSNC:
      TCOC_LSC_RETRIEVAL_REQ_FSNC();
```

```
        break;

    case MSG_MGMT_LSC_POWER_ON:
        MGMT_LSC_POWER_ON();
        break;

    case MSG_MGMT_LSC_LEVEL3_FAILURE:
        MGMT_LSC_LEVEL3_FAILURE();
        break;

    case MSG_MGMT_LSC_LOCAL_PRO_OUTAGE:
        MGMT_LSC_LOCAL_PRO_OUTAGE();
        break;

    case MSG_MGMT_LSC_LOCAL_PRO_RECOVERED:
        MGMT_LSC_LOCAL_PRO_RECOVERED();
        break;

    case MSG_XXX_RC_CONGESTION_DISCARD:
        XXX_RC_CONGESTION_DISCARD();
        break;

    case MSG_XXX_RC_CONGESTION_ACCEPT:
        XXX_RC_CONGESTION_DISCARD();
        break;

    case MSG_XXX_RC_NO_CONGESTION:
        XXX_RC_NO_CONGESTION();
        break;
}
```

G. EXECUTIVE FOR "TXC_rcv" STATE

```
/* service the interrupt by acquiring the arriving packet */
/* multiple arriving streams are supported. */

pkptr = op_pk_get (op_intrpt_strm ());


/* attempt to enqueue the packet at tail */
/* of subqueue 0.                        */
if (dv_TXC_state = STATE_TXC_IN_SERVICE) {
   if (op_subq_pk_insert (TRANSFER_BUFFER, pkptr, OPC_QPOS_TAIL) ==
OPC_QINS_OK) {
     dv_TXC_Cm = dv_TXC_Cm + 1;
   }
   else {

      /* the inserton failed (due to to a       */
      /* full queue) deallocate the packet. */
      op_pk_destroy (pkptr);
   }
}
else {
   /* TXC not ready yet, discard packet */
   op_pk_destroy(pkptr);
}
```

H. EXECUTIVE FOR "TX_MSU" STATE

```
/* ================================================================== */
/* Basic Transmission Control (TXC)                                   */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* Note:                                                              */
/*    This defines the processing of MSUs from layer 4 and also the   */
/*    creation of Link Status Signal Units and Fill In Status Units   */
/*    as defined in ITU-T Q.703 (Jul 96), Fig 13.                     */
/* ================================================================== */

/* Unlike standard SS7 where transmission requests are send by  */
/* the DAEDT function block. "trans_req" is set                 */
/* whenever a transmission is needed completed.                 */
/* LSSUs has the highest priority, followed by MSUs from RTB,   */
/* MSUs from TB and when there is nothing FISU is send          */
/* This means that whenever transmission is request, a SU is    */
/* always sent                                                  */

if ((trans_req) && (dv_TXC_state == STATE_TXC_IN_SERVICE)) {

   /* new transmission is requested again. */
   trans_req = FALSE;
}

if (dv_TXC_state == STATE_TXC_IN_SERVICE) {

   /* Send LSSU first whenever there is one, else try to send a MSU */
   if (dv_TXC_LSSU_available_flag) {

      /* set SU type to send to LSSU */
      SU_type_to_send = LSSU;

      /* stop future LSSU transmission is link is busy */
      if (dv_TXC_status_indication == LINK_STATUS_BUSY) {
         dv_TXC_LSSU_available_flag = FALSE;
      }
   }
   else {

      /* try to send MSU either from RTB or TB unless it is inhibited */
      if (dv_TXC_MSU_inhibited_flag) {

         /* set SU type to send to FISU */
         SU_type_to_send = FISU;
      }
      else {

         /* FSNT is always the same as FSNL unless it is changed by a NACK. */
         /* if a negative acknowledgement is received making FSNT not equal */
         /* to FSNL, send MSUs from RTB and keep doing it for next request  */
         /* till all has been sent                                          */
         if (dv_TXC_FSNT == dv_TXC_FSNL) {

            /* try to send MSU from TB.                */
            /* Send FISU if it is empty or RTB is full */
            if ((dv_TXC_Cm == 0) || (dv_TXC_RTB_full_flag)) {

               /* set SU type to send to FISU */
               SU_type_to_send = FISU;
            }
```

```
            else {

               /* fetch MSU pdu from TB */
               pkptr_tx_pdu = op_subq_pk_remove(TRANSFER_BUFFER,
OPC_QPOS_HEAD);

               /* decrement counters and incrment sequence */
               /* numbers since this is a new MSU          */
               dv_TXC_Cm = dv_TXC_Cm - 1;
               dv_TXC_FSNL = mtp2_increment(dv_TXC_FSNL, 1, MOD_SN);
               dv_TXC_FSNT = dv_TXC_FSNL;

               /* if this is the 1st MSU */
               if (dv_TXC_FSNL == dv_TXC_FSNF) {
                  mtp2_timer_start(&dv_timer7);
               }

               /* store MSU pdu to RTB */
               op_subq_pk_insert(RETRANSMISSION_BUFFER,
op_pk_copy(pkptr_tx_pdu), OPC_QPOS_TAIL);

               /* send the FSNT value to receiver control */
               TXC_RC_FSNT(dv_TXC_FSNT);

               /* set RTB full flag if RTB is going to be */
               /* full after this transmission            */
               if (dv_TXC_FSNL == mtp2_decrement(dv_TXC_FSNF, 2, MOD_SN)) {
                  dv_TXC_RTB_full_flag = TRUE;
               }

               /* set SU type to send to MSU */
               SU_type_to_send = MSU;
            }
         }
         else {

            /* send MSU from RTB */
            dv_TXC_FSNT = mtp2_increment(dv_TXC_FSNT, 1, MOD_SN);

            /* fetch MSU pdu from RTB */
            pkptr_tx_pdu = op_subq_pk_remove(RETRANSMISSION_BUFFER,
OPC_QPOS_HEAD);
            /* send the FSNT value to receiver control */
            TXC_RC_FSNT(dv_TXC_FSNT);
            /* set SU type to send to MSU */
            SU_type_to_send = MSU;
         }
      }
   }

   /* assign BSNT, BIBT and FIBT */
   dv_TXC_BSNT = mtp2_decrement(dv_TXC_FSNX, 1, MOD_SN);
   dv_TXC_BIBT = dv_TXC_BIB;
   dv_TXC_FIBT = dv_TXC_FIB;

   /* create the SU */
   pkptr_tx = op_pk_create(0);
   op_pk_fd_set(pkptr_tx, FD_INDEX_1ST, OPC_FIELD_TYPE_INTEGER, FLAG,
FD_LEN_FLAG);
   op_pk_fd_set(pkptr_tx, FD_INDEX_2ND, OPC_FIELD_TYPE_INTEGER, dv_TXC_BSNT,
FD_LEN_BSN);
```

```
   op_pk_fd_set(pkptr_tx, FD_INDEX_3RD, OPC_FIELD_TYPE_INTEGER, dv_TXC_BIBT,
FD_LEN_BIB);
   op_pk_fd_set(pkptr_tx, FD_INDEX_4TH, OPC_FIELD_TYPE_INTEGER, dv_TXC_FSNT,
FD_LEN_FSN);
   op_pk_fd_set(pkptr_tx, FD_INDEX_5TH, OPC_FIELD_TYPE_INTEGER, dv_TXC_FIBT,
FD_LEN_FIB);
   switch (SU_type_to_send) {
      case FISU:
         /* form FISU */
         op_pk_fd_set(pkptr_tx, FD_INDEX_6TH, OPC_FIELD_TYPE_INTEGER, 0,
FD_LEN_LI);
         op_pk_fd_set(pkptr_tx, FD_INDEX_7TH, OPC_FIELD_TYPE_INTEGER, 0,
FD_LEN_CK);
         op_pk_fd_set(pkptr_tx, FD_INDEX_8TH, OPC_FIELD_TYPE_INTEGER, FLAG,
FD_LEN_FLAG);
         break;

      case LSSU:
         /* form LSSU */
         i = FD_LEN_SF/8;
         op_pk_fd_set(pkptr_tx, FD_INDEX_6TH, OPC_FIELD_TYPE_INTEGER, i,
FD_LEN_LI);
         op_pk_fd_set(pkptr_tx, FD_INDEX_7TH, OPC_FIELD_TYPE_INTEGER,
dv_TXC_status_indication, FD_LEN_SF);
         op_pk_fd_set(pkptr_tx, FD_INDEX_8TH, OPC_FIELD_TYPE_INTEGER, 0,
FD_LEN_CK);
         op_pk_fd_set(pkptr_tx, FD_INDEX_9TH, OPC_FIELD_TYPE_INTEGER, FLAG,
FD_LEN_FLAG);
         break;

      case MSU:
         /* form MSU */
         i = op_pk_total_size_get(pkptr_tx_pdu)/8;
         op_pk_fd_set(pkptr_tx, FD_INDEX_6TH, OPC_FIELD_TYPE_INTEGER, i,
FD_LEN_LI);
         op_pk_fd_set(pkptr_tx, FD_INDEX_7TH, OPC_FIELD_TYPE_PACKET,
pkptr_tx_pdu, FD_LEN_PDU);
         op_pk_fd_set(pkptr_tx, FD_INDEX_8TH, OPC_FIELD_TYPE_INTEGER, 0,
FD_LEN_CK);
         op_pk_fd_set(pkptr_tx, FD_INDEX_9TH, OPC_FIELD_TYPE_INTEGER, FLAG,
FD_LEN_FLAG);
         break;
   }

   /* determine the packets length (in bits)  */
   SU_len = op_pk_total_size_get(pkptr_tx);
   /* determine the time required to complete */
   /* transmission of the packet               */
   SU_tx_time = SU_len / service_rate;

   /* schedule an interrupt for this process  */
   /* to simluate this transmission delay     */
   op_intrpt_schedule_self (op_sim_time() + SU_tx_time, TXC_TRANS_COMPLETE);

   /* activate the transmission flag to ensure */
   /* that SU is sent at end of delay */
   trans_req = TRUE;

   /* forward the copied packet on stream 0, causing */
   /* an immediate interrupt at destination.        */
   op_pk_send(pkptr_tx, STREAM_MTP2_LOW_LEVEL_OUT);
}
```

I. EXECUTIVE FOR "TXC" STATE

```
/* ================================================================== */
/* Basic Transmission Control (TXC)                                   */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* Note:                                                              */
/*    This defines all TXC processing which includes handling of      */
/*    incoming internal messages and timer expiry.                    */
/*    Messages are proccessed in FIFO order.                          */
/*    Each message is processed according to the algorithm            */
/*    as defined in ITU-T Q.703 (Jul 96), Fig 13.                     */
/* ================================================================== */


switch (op_intrpt_code()) {
   case TXC_MSG:

      /* If incoming message exists, remove the message packet */
      /* from the buffer and process it                        */
      /* The first field in each packet is always the message  */
      /* code, it can be followed by one or more fields        */
      /* depending on the message                              */

      if (!op_subq_empty(TXC_BUFFER)) {
         pkptr = op_subq_pk_remove(TXC_BUFFER, OPC_QPOS_HEAD);
         op_pk_fd_get(pkptr, FD_INDEX_1ST, &msg_code);
         switch (msg_code) {

            case MSG_LSC_TXC_START:
               if (dv_TXC_state == STATE_TXC_IDLE) {
                  dv_TXC_LSSU_available_flag = FALSE;
                  dv_TXC_SIB_received_flag = FALSE;
                  dv_TXC_RTB_full_flag = FALSE;
                  dv_TXC_MSU_inhibited_flag = FALSE;
                  dv_TXC_FSNL = 127;
                  dv_TXC_FSNT = 127;
                  dv_TXC_FSNX = 0;
                  dv_TXC_FSNF = 0;
                  dv_TXC_FIB  = 1;
                  dv_TXC_BIB  = 1;
                  dv_TXC_Cm   = 0;
                  op_subq_flush(TRANSFER_BUFFER);
                  op_subq_flush(RETRANSMISSION_BUFFER);
                  dv_TXC_state = STATE_TXC_IN_SERVICE;
                  op_prg_log_entry_write(log_hdle_link_status, "TXC: state
IDLE->IN_SERVICE, instructed by LSC");
               }
               else if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
                  dv_TXC_SIB_received_flag = FALSE;
                  dv_TXC_RTB_full_flag = FALSE;
                  dv_TXC_MSU_inhibited_flag = FALSE;
                  dv_TXC_FSNL = 127;
                  dv_TXC_FSNT = 127;
                  dv_TXC_FSNX = 0;
                  dv_TXC_FSNF = 0;
                  dv_TXC_FIB  = 1;
                  dv_TXC_BIB  = 1;
                  dv_TXC_Cm   = 0;
                  op_subq_flush(TRANSFER_BUFFER);
                  op_subq_flush(RETRANSMISSION_BUFFER);
               }
```

```
            break;

         case MSG_LSC_TXC_SEND_SIOS:

            /* T7, timer for excessive acknowledgement is not needed */
            /* as these SUs do not required acknowledgements         */
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               mtp2_timer_stop(&dv_timer7);
               dv_TXC_LSSU_available_flag = TRUE;
               dv_TXC_status_indication = LINK_STATUS_OUT_SERVICE;
               op_prg_log_entry_write(log_hdle_link_status, "TXC: Sending
SIOS, instructed by LSC");
            }
            break;

         case MSG_LSC_TXC_SEND_SIPO:

            /* T7, timer for excessive acknowledgement is not needed */
            /* as these SUs do not required acknowledgements         */
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               mtp2_timer_stop(&dv_timer7);
               dv_TXC_LSSU_available_flag = TRUE;
               dv_TXC_status_indication = LINK_STATUS_PROCESSOR_OUTAGE;
               op_prg_log_entry_write(log_hdle_link_status, "TXC: Sending
SIPO, instructed by LSC");
            }
            break;

         case MSG_IAC_TXC_SEND_SIO:
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               dv_TXC_LSSU_available_flag = TRUE;
               dv_TXC_status_indication = LINK_STATUS_OUT_ALIGNMENT;
               op_prg_log_entry_write(log_hdle_link_status, "TXC: Sending
SIO, instructed by IAC");
            }
            break;

         case MSG_IAC_TXC_SEND_SIN:
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               dv_TXC_LSSU_available_flag = TRUE;
               dv_TXC_status_indication = LINK_STATUS_NORMAL;
               op_prg_log_entry_write(log_hdle_link_status, "TXC: Sending
SIN, instructed by IAC");
            }
            break;

         case MSG_IAC_TXC_SEND_SIE:
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               dv_TXC_LSSU_available_flag = TRUE;
               dv_TXC_status_indication = LINK_STATUS_EMERGENCY;
               op_prg_log_entry_write(log_hdle_link_status, "TXC: Sending
SIE, instructed by IAC");
            }
            break;

         case MSG_CC_TXC_SEND_SIB:
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               dv_TXC_LSSU_available_flag = TRUE;
               dv_TXC_status_indication = LINK_STATUS_BUSY;
               op_prg_log_entry_write(log_hdle_link_status, "TXC: Sending
SIB, instructed by CC");
             }
```

```
            break;

         case MSG_LSC_TXC_SEND_FISU:

            /* T7, timer for excessive acknowledgement is not needed */
            /* as these SUs do not required acknowledgements         */
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               mtp2_timer_stop(&dv_timer7);
               dv_TXC_MSU_inhibited_flag = TRUE;
               dv_TXC_LSSU_available_flag = FALSE;
               op_prg_log_entry_write(log_hdle_link_status, "TXC: Sending
FISU only, instructed by LSC");
            }
            break;

         case MSG_LSC_TXC_SEND_MSU:

            /* transmit MSU from transfer buffer if RTB is empty */
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {

               /* check whether RTB is empty */
               if (dv_TXC_FSNL != mtp2_decrement(dv_TXC_FSNF, 1, MOD_SN)) {
                  mtp2_timer_start(&dv_timer7);
               }
               dv_TXC_MSU_inhibited_flag = FALSE;
               dv_TXC_LSSU_available_flag = FALSE;
               op_prg_log_entry_write(log_hdle_link_status, "TXC: Sending
MSU if any, instructed by LSC");

               /* init statistic calculation variables */
               start_time = op_sim_time();
               total_bits_rcv = 0;
            }
            break;

         case MSG_RC_TXC_SEND_NACK:

            /* send NACK by inverting the BIB bit */
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               if (dv_TXC_BIB == 1) {
                  dv_TXC_BIB = 0;
               }
               else {
                  dv_TXC_BIB = 1;
               }
               op_prg_log_entry_write(log_hdle_link_status, "TXC: Sending
NACK, instructed by RC");
            }
            break;

         case MSG_RC_TXC_SIB_RECEIVED:

            /* the receiver RC has receoved SIB from remote SP    */
            /* if this is the first time, start congestion timer, */
            /* T6 else start T7, excessive acknowledgment timer   */
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               if (!dv_TXC_SIB_received_flag) {
                  mtp2_timer_start(&dv_timer6);
                  dv_TXC_SIB_received_flag = TRUE;
               }
               mtp2_timer_start(&dv_timer7);
            }
```

```
            break;

         case MSG_RC_TXC_BSNR_AND_BIBR:
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               op_pk_fd_get(pkptr, FD_INDEX_2ND, &dv_TXC_BSNR);
               op_pk_fd_get(pkptr, FD_INDEX_3RD, &dv_TXC_BIBR);

               /* BSNR gives the value of the last acknowledged MSU      */
               /* If 1st MSU in the RTB is not the last acknowledged MSU */
               if (dv_TXC_FSNF != mtp2_increment(dv_TXC_BSNR, 1, MOD_SN)) {

                /* stop congestion timer since new MSUs has been received */
                  if (dv_TXC_SIB_received_flag) {
                     dv_TXC_SIB_received_flag = FALSE;
                     mtp2_timer_stop(&dv_timer6);
                  }

                  /* erase in RTB, MSU's from latest up to FSN = BSNR */
                  /* i.e. stop when FSNF = BSNR + 1                   */
                  while (dv_TXC_FSNF != mtp2_increment(dv_TXC_BSNR, 1,
MOD_SN)) {
                     if (!op_subq_empty(RETRANSMISSION_BUFFER)) {
                        op_subq_pk_remove(RETRANSMISSION_BUFFER,
OPC_QPOS_HEAD);
                     }
                     dv_TXC_FSNF = mtp2_increment(dv_TXC_FSNF, 1, MOD_SN);
                  }

                  /* if RTB is empty */
                  if (dv_TXC_FSNL != mtp2_decrement(dv_TXC_FSNF, 1, MOD_SN))
{
                     mtp2_timer_start(&dv_timer7);
                  }
                  else {
                     mtp2_timer_stop(&dv_timer7);
                  }
                  dv_TXC_RTB_full_flag = FALSE;
               }

               /* if negative acknowledgment has been received         */
               /* shift FSNT counter to activate a retransmiision loop */
               /* during next transmission cycle                       */
               /* First transmitted MSU from RTB will be FSNF          */
               if (dv_TXC_FIB != dv_TXC_BIBR) {

                /* stop congestion timer since new MSUs has been received */
                  if (dv_TXC_SIB_received_flag) {
                     dv_TXC_SIB_received_flag = FALSE;
                     mtp2_timer_stop(&dv_timer6);
                  }
                  dv_TXC_FIB = dv_TXC_BIBR;
                  dv_TXC_FSNT = dv_TXC_FSNF - 1;
               }
            }
            break;

         case MSG_RC_TXC_FSNX:
            if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
               op_pk_fd_get(pkptr, FD_INDEX_2ND, &dv_TXC_FSNX);
            }
            break;
```

```
            case MSG_LSC_TXC_RETRIEVAL_REQ_FSNC:
               if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
                  op_pk_fd_get(pkptr, FD_INDEX_2ND, &dv_TXC_FSNC);

                  /* erase in RTB MSUs up to FSN = FSNC */
                  while (dv_TXC_FSNF != mtp2_increment(dv_TXC_FSNC, 1, MOD_SN))
{
                     if (!op_subq_empty(RETRANSMISSION_BUFFER)) {
                        op_subq_pk_remove(RETRANSMISSION_BUFFER,
OPC_QPOS_HEAD);
                     }
                     dv_TXC_FSNF = mtp2_increment(dv_TXC_FSNF, 1, MOD_SN);
                  }

                  /* send Msu messages from RTB to level 3 */
                  /* from FSN = FSNF up to FSN = FSNL      */
                  while (dv_TXC_FSNF != mtp2_increment(dv_TXC_FSNL, 1, MOD_SN))
{
                     if (!op_subq_empty(RETRANSMISSION_BUFFER)) {
                        op_pk_deliver(op_subq_pk_remove(RETRANSMISSION_BUFFER,
OPC_QPOS_HEAD), mtp3_id, mtp3_retrieve_input_stream);
                     }
                     dv_TXC_FSNF = mtp2_increment(dv_TXC_FSNF, 1, MOD_SN);
                  }

                  /* send Msu messages from TB to level 3 */
                  while (dv_TXC_Cm != 0) {
                     if (!op_subq_empty(TRANSFER_BUFFER)) {
                        op_pk_deliver(op_subq_pk_remove(TRANSFER_BUFFER,
OPC_QPOS_HEAD), mtp3_id, mtp3_retrieve_input_stream);
                     }
                     dv_TXC_Cm = dv_TXC_Cm - 1;
                  }
                  dv_TXC_RTB_full_flag = FALSE;

                  /* send retrieval complete */
                  mtp2_send_external_msg(mtp3_id,
MSG_TXC_TCOC_RETRIEVAL_COMPLETE, 0, iciptr_external);
                  dv_TXC_FSNL = dv_TXC_FSNC;
                  dv_TXC_FSNT = dv_TXC_FSNL;
                  op_prg_log_entry_write(log_hdle_link_status, "TXC: MSU
retrieval, as instructed by LSC");
               }
               break;

            case MSG_LSC_TXC_FLUSH_BUFFERS:
               if (dv_TXC_state == STATE_TXC_IN_SERVICE) {

                  /* clear RTB and TB */
                  op_subq_flush(RETRANSMISSION_BUFFER);
                  op_subq_flush(TRANSFER_BUFFER);

                  dv_TXC_RTB_full_flag = FALSE;
                  dv_TXC_Cm    = 0;
                  dv_TXC_FSNF = mtp2_increment(dv_TXC_BSNR, 1, MOD_SN);
                  dv_TXC_FSNL = dv_TXC_BSNR;
                  dv_TXC_FSNT = dv_TXC_BSNR;
                  mtp2_timer_stop(&dv_timer7);
                  op_prg_log_entry_write(log_hdle_link_status, "TXC: Buffers
Flushed, as instructed by LSC");
               }
```

```
                break;
            }
            op_pk_destroy(pkptr);
        }


        /* schedule next internal message processing interrupt */
        op_intrpt_schedule_self(op_sim_time() + TXC_service_time, TXC_MSG);
        break;

    case T6_EXPIRE:

        /* Process expiry of T6, Remote Congestion Timer */
        if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
            TXC_LSC_LINK_FAILURE();
            dv_TXC_SIB_received_flag = FALSE;
            mtp2_timer_stop(&dv_timer7);
            op_prg_log_entry_write(log_hdle_link_status, "TXC: T6 Expire");
        }
        break;

    case T7_EXPIRE:

        /* Process expiry of T7, Excessive delay in Acknowldegement Timer */
        if (dv_TXC_state == STATE_TXC_IN_SERVICE) {
            TXC_LSC_LINK_FAILURE();
            mtp2_timer_stop(&dv_timer6);
            dv_TXC_SIB_received_flag = FALSE;
            dv_TXC_state = STATE_TXC_IN_SERVICE;
            op_prg_log_entry_write(log_hdle_link_status, "TXC: T7 Expire, excess
delay in ACK");
        }
        break;
}
```

J. EXECUTIVE FOR "RC_rcv" STATE

```
/* ================================================================== */
/* SU receiving state for Basic Reception Control (RC)                */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* Note:                                                              */
/* This process does the following tasks:                             */
/* (1) Retrieve the packet and clear in from the input stream.        */
/* (2) Allocate errors to the incoming packets to simulate error      */
/*     occuring in a SS7 link. Zero Error models are assume in the    */
/*     pipeline stages.                                               */
/* (3) If the packet is deemed to be correct, each message are        */
/*     processed according to the algorithm as defined in             */
/*     ITU-T Q.703 (Jul 96), Fig 14. Basic Reception Control          */
/* ================================================================== */

/* ================================================================== */
/* (1) Retrieving packet from input stream                            */
/* ================================================================== */
pkptr_rx = op_pk_get(op_intrpt_strm ());

/* ================================================================== */
/* (2) Allocate link errors                                           */
/* ================================================================== */

/* zero error algorithm used at the moment */
/* This two function can be modified to simulate non error free links */
packet_no_error = DAEDR(pkptr_rx);

/* ================================================================== */
/* (3) Process packet if packet is deemed to be error free from DAEDR */
/* ================================================================== */

/* processed the packet according to the state and packet type */
if (packet_no_error) {
   switch (dv_RC_state) {
      case STATE_RC_IDLE:
         op_pk_destroy(pkptr_rx);
         break;

      case STATE_RC_IN_SERVICE:

         /* get the header fields from the packet and destroys it */
         op_pk_fd_get(pkptr_rx, FD_INDEX_2ND, &dv_RC_BSNR);
         op_pk_fd_get(pkptr_rx, FD_INDEX_3RD, &dv_RC_BIBR);
         op_pk_fd_get(pkptr_rx, FD_INDEX_4TH, &dv_RC_FSNR);
         op_pk_fd_get(pkptr_rx, FD_INDEX_5TH, &dv_RC_FIBR);
         op_pk_fd_get(pkptr_rx, FD_INDEX_6TH, &dv_RC_LI);

         /* LSSU has LI equals to 1 or 2, MSU has LI > 2 */
         if ((dv_RC_LI == 1) || (dv_RC_LI == 2)) {
            op_pk_fd_get(pkptr_rx, FD_INDEX_7TH, &dv_RC_status_indication);
         }
         else if (dv_RC_LI > 2) {
            op_pk_fd_get(pkptr_rx, FD_INDEX_7TH, &pkptr_rx_pdu);
         }
         op_pk_destroy(pkptr_rx);

         /* if Signal Unit received is a LSSU */
         if ((dv_RC_LI == 1) || (dv_RC_LI == 2)) {
```

```
      switch (dv_RC_status_indication) {
         case LINK_STATUS_NORMAL:
            RC_LSC_SIN   ();
            RC_IAC_SIN   ();
            break;

         case LINK_STATUS_EMERGENCY:
            RC_LSC_SIE();
            RC_IAC_SIE();
            break;

         case LINK_STATUS_OUT_ALIGNMENT:
            RC_LSC_SIO();
            RC_IAC_SIO();
            break;

         case LINK_STATUS_OUT_SERVICE:
            RC_LSC_SIOS();
            RC_IAC_SIOS();
            break;

         case LINK_STATUS_PROCESSOR_OUTAGE:
            RC_LSC_SIPO();
            break;

         case LINK_STATUS_BUSY:
            RC_TXC_SIB_RECEIVED();
            break;
      }
   }
   else {

      /* check for valid BSNR */
      if (is_BSNR_valid(dv_RC_FSNF, dv_RC_FSNT, dv_RC_BSNR)) {

         /* valid BSNR received, if previous SU has abnormal BSNR    */
         /* mark this as a first time recovery, but still discard SU */
         if (dv_RC_abnormal_BSNR_flag && (dv_RC_UNB != 1)) {
            dv_RC_UNB = 1;
         }
         else {

            /* valid BSNR received, if previous SU has abnormal BSNR  */
            /* and this is 2nd time recovery, confirm recovery, reset */
            /* abnormal BSNR flag and continue to check th SU         */
            if (dv_RC_abnormal_BSNR_flag && (dv_RC_UNB ==1)) {
               dv_RC_abnormal_BSNR_flag = FALSE;
            }
            if (dv_RC_FIBR = dv_RC_FIBX) {

               /* normal FIBR detected, if previous SU has abnormal  */
               /* FIBR set recovered from abnormal FIBR, ie. UNF to  */
               /* 1. but still diacard the signal unit. If this is   */
               /* the second correct FIBR after an abnormal FIBR,    */
               /* confirm that this is good recovery and continue to */
               /* process SU                                         */
               if ((dv_RC_abnormal_FIBR_flag) && (dv_RC_UNF != 1)) {

                  /* 1st time recovery from abnormal FIBR, */
                  /* still reject the SU                   */
                  dv_RC_UNF = 1;
```

```
                    }
                    else {

                       /* 2nd time recovery, clear abnormal FIBR flag */
                       if ((dv_RC_abnormal_FIBR_flag) && (dv_RC_UNF == 1)) {
                          dv_RC_abnormal_FIBR_flag = FALSE;
                       }

                       RC_LSC_FISU_MSU_RECEIVED();
                       RC_TXC_BSNR_AND_BIBR(dv_RC_BSNR, dv_RC_BIBR);
                       dv_RC_FSNF = mtp2_increment(dv_RC_BSNR, 1, MOD_SN);

                       /* process the MSU/FISU if RC is in accept MSU/FISU */
                       /* mode as controlled by LSC, else discard it       */
                       if (dv_RC_MSU_FISU_accepted_flag) {

                          /* if RC is in congestion discard mode discard   */
                          /* the SU but mark it for retransmission check   */
                          /* by TXC                                        */
                          if (dv_RC_congestion_discard_flag) {
                             if (dv_RC_LI > 2) {
                                dv_RC_RTR = 1;
                             }
                             RC_CC_BUSY();
                          }
                          else {

                             /* check for valid MSU */
                             if (dv_RC_FSNR == dv_RC_FSNX) {
                                if (dv_RC_LI > 2) {

                                   /* valid MSU, forward it to level 3 */
                                   total_bits_rcv = total_bits_rcv +
op_pk_total_size_get(pkptr_rx_pdu) + 32;
                                   op_stat_write(link_utilization_handle,
total_bits_rcv/(op_sim_time() - start_time));
                                   op_pk_send(pkptr_rx_pdu,
STREAM_MTP2_HIGH_LEVEL_OUT);
                                   dv_RC_FSNX = mtp2_increment(dv_RC_FSNX, 1,
MOD_SN);
                                   dv_RC_RTR = 0;
                                   if (dv_RC_congestion_accept_flag) {
                                      RC_CC_BUSY();
                                   }
                                   else {
                                      RC_TXC_FSNX(dv_RC_FSNX);
                                   }
                                }
                                else {

                                   /* invalid SU, ask TXC to send NACK */
                                   if (dv_RC_congestion_accept_flag) {
                                      RC_CC_BUSY();
                                   }
                                   else {
                                      RC_TXC_SEND_NACK();
                                      dv_RC_RTR = 1;

                                      /* invert FIBX since own TXC will  */
                                      /* invert BIB and the remote TXC   */
                                      /* when will used the new BIB as   */
                                      /* FIB, when it receives the NACK  */
```

```
                                    if (dv_RC_FIBX == 1) {
                                       dv_RC_FIBX = 0;
                                    }
                                    else {
                                       dv_RC_FIBX = 1;
                                    }
                                 }
                              }
                           }
                           else if (dv_RC_FSNR != mtp2_decrement(dv_RC_FSNX,
1, MOD_SN)) {

                              /* invalid FISU */
                              /* get TXC to send NACK since this is not a
valid FISU */

                              if (dv_RC_congestion_accept_flag) {
                                 RC_CC_BUSY();
                              }
                              else {
                                 RC_TXC_SEND_NACK();
                                 dv_RC_RTR = 1;
                                 if ( dv_RC_FIBX == 1) {
                                    dv_RC_FIBX = 0;
                                 }
                                 else {
                                    dv_RC_FIBX = 1;
                                 }
                              }
                           }
                        }
                     }
                  }
                  else {

                     /* abnormal FIBR detected, discard the SU */
                     if (dv_RC_abnormal_FIBR_flag) {

                        /* declare link failure as this is 2nd consecutive */
                        /* time abnormal FIBR                              */
                        RC_LSC_LINK_FAILURE();
                        dv_RC_state = STATE_RC_IDLE;
                        op_prg_log_entry_write(
                           log_hdle_link_status,
                           "RC: 2nd abnormal FIBR, FIBX: %d, FIBR: %d",
                           dv_RC_FIBX,
                           dv_RC_FIBR);
                        op_prg_log_entry_write(
                           log_hdle_link_status,
                           "RC: state IN_SERVICE->IDLE, link failure due to
abnormal FIBR twice");
                     }
                     else {

                        /* first time abnormal FIBR, if retransmission is */
                        /* required, send BSNR and BIBR updates to TXC,   */
                        /* else mark abnormal FIBR, UNF, counter for      */
                        /* unreasonable FIBR is started by reset          */
                        if (dv_RC_RTR == 1) {
                           RC_TXC_BSNR_AND_BIBR(dv_RC_BSNR, dv_TXC_BIBR);
                           dv_RC_FSNF = mtp2_increment(dv_RC_BSNR, 1, MOD_SN);
```

```
                        }
                        else {
                           dv_RC_abnormal_FIBR_flag = TRUE;
                           dv_RC_UNF = 0;
                           op_prg_log_entry_write(
                              log_hdle_link_status,
                              "RC: 1st abnormal FIBR, FIBX: %d, FIBR: %d",
                              dv_RC_FIBX,
                              dv_RC_FIBR);
                        }
                     }
                  }
               }
               else {

                  /* abnormal BSNR detected for current SU, if this is the 2nd */
                  /* time, declare link failure                                 */
                  if (dv_RC_abnormal_BSNR_flag) {
                     RC_LSC_LINK_FAILURE();
                     dv_RC_state = STATE_RC_IDLE;
                     op_prg_log_entry_write(
                        log_hdle_link_status,
                        "RC: 2nd abnormal BSNR, FSNF: %d, FSNT: %d, BSNR: %d",
                        dv_RC_FSNF,
                        dv_RC_FSNT,
                        dv_RC_BSNR);
                     op_prg_log_entry_write(
                        log_hdle_link_status,
                        "RC: state IN_SERVICE->IDLE, link failure due to abnormal
BSNR twice");
                  }
                  else {
                     dv_RC_abnormal_BSNR_flag = TRUE;
                     dv_RC_UNB = 0;
                     op_prg_log_entry_write(
                        log_hdle_link_status,
                        "RC: 1st abnormal BSNR, FSNF: %d, FSNT: %d, BSNR: %d",
                        dv_RC_FSNF,
                        dv_RC_FSNT,
                        dv_RC_BSNR);
                  }
               }
            }
            break;
      }
}
```

K. EXECUTIVE FOR "RC" STATE

```
/* ========================================================================== */
/* Basic Reception Control (RC)                                               */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)             */
/* Note:                                                                      */
/*    This defines all RC processing which includes handling of               */
/*    incoming internal messages and timer expiry.                            */
/*    Messages are proccessed in FIFO order.                                  */
/*    Each message is processed according to the algorithm                    */
/*    as defined in ITU-T Q.703 (Jul 96), Fig 14.                             */
/* ========================================================================== */


/* If incoming message exists, remove the message packet */
/* from the buffer and process it                        */
/* The first field in each packet is always the message  */
/* code, it can be followed by one or more fields        */
/* depending on the message                              */

if (!op_subq_empty(RC_BUFFER)) {
   pkptr = op_subq_pk_remove(RC_BUFFER, OPC_QPOS_HEAD);
   op_pk_fd_get(pkptr, FD_INDEX_1ST, &msg_code);
   switch (msg_code) {

      case MSG_LSC_RC_START:
         if (dv_RC_state == STATE_RC_IDLE) {
            dv_RC_FSNX = 0;
            dv_RC_FIBX = 1;
            dv_RC_FSNF = 0;
            dv_RC_FSNT = 127;
            dv_RC_RTR  = 0;
            dv_RC_MSU_FISU_accepted_flag  = FALSE;
            dv_RC_abnormal_BSNR_flag      = FALSE;
            dv_RC_abnormal_FIBR_flag      = FALSE;
            dv_RC_congestion_discard_flag = FALSE;
            dv_RC_congestion_accept_flag  = FALSE;
            dv_RC_state = STATE_RC_IN_SERVICE;
            op_prg_log_entry_write(log_hdle_link_status, "RC: state IDLE-
>IN_SERVICE, START as instructed by LSC");
         }
         break;

      case MSG_LSC_RC_STOP:
         if (dv_RC_state == STATE_RC_IN_SERVICE) {
            RC_CC_NORMAL();
            dv_RC_state = STATE_RC_IDLE;
            op_prg_log_entry_write(log_hdle_link_status, "RC: state IN_SERVICE-
>IDLE, STOP as instructed by LSC");
         }
         break;

      case MSG_LSC_RC_RETRIEVE_BSNT:
         if ((dv_RC_state == STATE_RC_IDLE) || (dv_RC_state ==
STATE_RC_IN_SERVICE)) {
            dv_RC_BSNT = mtp2_decrement(dv_RC_FSNX, 1, MOD_SN);

            /* send to L3 BSNT number */
            mtp2_send_external_msg(mtp3_id, MSG_RC_TCOC_BSNT, dv_RC_BSNT,
iciptr_external);
         }
```

```
        break;

    case MSG_LSC_RC_RETRIEVE_FSNX:
        if (dv_RC_state == STATE_RC_IN_SERVICE) {
            RC_TXC_FSNX(dv_RC_FSNX);
            dv_RC_congestion_discard_flag = FALSE;
            dv_RC_congestion_accept_flag  = FALSE;
            RC_CC_NORMAL();
            dv_RC_RTR = 0;
        }
        break;

    case MSG_LSC_RC_REJECT_MSU_FISU:
        if (dv_RC_state == STATE_RC_IN_SERVICE) {
            dv_RC_MSU_FISU_accepted_flag = FALSE;
        }
        break;

    case MSG_LSC_RC_ACCEPT_MSU_FISU:
        if (dv_RC_state == STATE_RC_IN_SERVICE) {
            dv_RC_MSU_FISU_accepted_flag = TRUE;
        }
        break;

    case MSG_TXC_RC_FSNT:
        if (dv_RC_state == STATE_RC_IN_SERVICE) {
            op_pk_fd_get(pkptr, FD_INDEX_2ND, &dv_RC_FSNT);
        }
        break;


    case MSG_XXX_RC_CONGESTION_DISCARD:
        if (dv_RC_state == STATE_RC_IN_SERVICE) {
            dv_RC_congestion_discard_flag = TRUE;
        }
        break;

    case MSG_XXX_RC_CONGESTION_ACCEPT:
        if (dv_RC_state == STATE_RC_IN_SERVICE) {
            dv_RC_congestion_accept_flag = TRUE;
        }
        break;

    case MSG_XXX_RC_NO_CONGESTION:
        if (dv_RC_state == STATE_RC_IN_SERVICE) {
            dv_RC_congestion_discard_flag = FALSE;
            dv_RC_congestion_accept_flag = FALSE;
            RC_CC_NORMAL();
            RC_TXC_FSNX(dv_RC_FSNX);
            if (dv_RC_RTR == 1) {
                RC_TXC_SEND_NACK();
                if (dv_RC_FIBX == 0) {
                    dv_RC_FIBX = 1;
                }
                else {
                    dv_RC_FIBX = 0;
                }
            }
        }
        break;
}
op_pk_destroy(pkptr);
```

```
}

/* schedule next internal message processing interrupt */
op_intrpt_schedule_self(op_sim_time() + RC_service_time, RC_MSG);
```

L. EXECUTIVE FOR "LSC" STATE

```
/* ================================================================== */
/* Link State Control (LSC)                                           */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* Note:                                                              */
/*    This defines all LSC processing which includes handling of      */
/*    incoming internal messages and timer expiry interrupts.         */
/*    Messages are proccessed in FIFO order.                          */
/*    Each message is processed according to the algorithm            */
/*    as defined in ITU-T Q.703 (Jul 96), Fig 8.                      */
/* ================================================================== */

switch (op_intrpt_code()) {
   case LSC_MSG:

      /* If incoming message exists, remove the message packet */
      /* from the buffer and process it                        */
      /* The first field in each packet is always the message  */
      /* code, it can be followed by one or more fields        */
      /* depending on the message                              */

      if (!op_subq_empty(LSC_BUFFER)) {
         pkptr = op_subq_pk_remove(LSC_BUFFER, OPC_QPOS_HEAD);
         op_pk_fd_get(pkptr, FD_INDEX_1ST, &msg_code);
         switch (msg_code) {

            case MSG_IAC_LSC_ALIGNMENT_COMPLETE:
               if (dv_LSC_state == STATE_LSC_INITIAL_ALIGNMENT) {
                  LSC_SUERM_START();
                  mtp2_timer_start(&dv_timer1);

                  /* go to "aligned not ready" state and wait for recovery if
local */
                  /* processor outage, else go ahead and assume "alignment
ready"   */
                  if (dv_LSC_local_processor_outage_during_alignment_flag) {
                     LSC_POC_LOCAL_PRO_OUTAGE();
                     LSC_TXC_SEND_SIPO();
                     LSC_RC_REJECT_MSU_FISU();
                     dv_LSC_state = STATE_LSC_ALIGNED_NOT_READY;
                     op_prg_log_entry_write(log_hdle_link_status, "LSC: state
INITIAL_ALIGNMENT->ALIGNED_NOT_READY");
                  }
                  else {
                     LSC_TXC_SEND_FISU();
                     LSC_RC_ACCEPT_MSU_FISU();
                     dv_LSC_state = STATE_LSC_ALIGNED_READY;
                     op_prg_log_entry_write(log_hdle_link_status, "LSC: state
INITIAL_ALIGNMENT->ALIGNED_READY");
                  }
               }
               break;

            case MSG_IAC_LSC_ALIGNMENT_NOT_POSSIBLE:

               /* alignment not possible, reset flags and go "out of service"
*/
               /* RC is stopped, IAC will stop by itself
*/
               if (dv_LSC_state == STATE_LSC_INITIAL_ALIGNMENT) {
```

```
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_RC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
INITIAL_ALIGNMENT->OUT_SERVICE");
             }
             break;

          case MSG_TXC_LSC_LINK_FAILURE:

             /* link failed, remote congestion detected by TXC    */
             /* processor outage flags need not be reset          */
             /* since it has to be clear for LSC to be in service */
             if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->OUT_SERVICE");
             }
             break;

          case MSG_RC_LSC_SIO:
             if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
                mtp2_timer_stop(&dv_timer1);
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_RC_STOP();
                LSC_SUERM_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_READY->OUT_SERVICE");
             }
             else if (dv_LSC_state == STATE_LSC_ALIGNED_NOT_READY) {
                mtp2_timer_stop(&dv_timer1);
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_RC_STOP();
                LSC_SUERM_STOP();
                LSC_TXC_SEND_SIOS();
                LSC_POC_STOP();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_NOT_READY->OUT_SERVICE");
             }
             else if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
```

```
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->OUT_SERVICE");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_POC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
PROCESSOR_OUTAGE->OUT_SERVICE");
             }
             break;

          case MSG_RC_LSC_SIN:
             if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->OUT_SERVICE");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_POC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
PROCESSOR_OUTAGE->OUT_SERVICE");
             }
             break;

          case MSG_RC_LSC_SIE:
             if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->OUT_SERVICE");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
```

```
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_POC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
PROCESSOR_OUTAGE->OUT_SERVICE");
             }
             break;

          case MSG_RC_LSC_SIOS:
             if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
                mtp2_timer_stop(&dv_timer1);
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_RC_STOP();
                LSC_SUERM_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_READY->OUT_SERVICE, SIOS received");
             }
             else if (dv_LSC_state == STATE_LSC_ALIGNED_NOT_READY) {
                mtp2_timer_stop(&dv_timer1);
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_RC_STOP();
                LSC_SUERM_STOP();
                LSC_TXC_SEND_SIOS();
                LSC_POC_STOP();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_NOT_READY->OUT_SERVICE, SIOS received");
             }
             else if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->OUT_SERVICE, SIOS received");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_POC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
```

```
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
PROCESSOR_OUTAGE->OUT_SERVICE, SIOS received");
             }
             break;

          case MSG_RC_LSC_SIPO:
             if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
                mtp2_timer_stop(&dv_timer1);
                mtp2_send_external_msg(mtp3_id,
MSG_LSC_LSAC_REMOTE_PRO_OUTAGE, own_id, iciptr_external);
                LSC_POC_REMOTE_PRO_OUTAGE();
                dv_LSC_state = STATE_LSC_PROCESSOR_OUTAGE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_READY->PROCESSOR_OUTAGE");
             }
             else if (dv_LSC_state == STATE_LSC_ALIGNED_NOT_READY) {
                mtp2_send_external_msg(mtp3_id,
MSG_LSC_LSAC_REMOTE_PRO_OUTAGE, own_id, iciptr_external);
                LSC_POC_REMOTE_PRO_OUTAGE();
                mtp2_timer_stop(&dv_timer1);
                dv_LSC_state = STATE_LSC_PROCESSOR_OUTAGE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_NOT_READY->PROCESSOR_OUTAGE");
             }
             else if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
                LSC_TXC_SEND_FISU();
                mtp2_send_external_msg(mtp3_id,
MSG_LSC_LSAC_REMOTE_PRO_OUTAGE, own_id, iciptr_external);
                LSC_POC_REMOTE_PRO_OUTAGE();
                dv_LSC_processor_outage_during_service_flag = TRUE;
                dv_LSC_state = STATE_LSC_PROCESSOR_OUTAGE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->PROCESSOR_OUTAGE");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                mtp2_send_external_msg(mtp3_id,
MSG_LSC_LSAC_REMOTE_PRO_OUTAGE, own_id, iciptr_external);
                LSC_POC_REMOTE_PRO_OUTAGE();
             }
             break;

          case MSG_RC_LSC_FISU_MSU_RECEIVED:
             if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_IN_SERVICE,
own_id, iciptr_external);
                mtp2_timer_stop(&dv_timer1);
                LSC_TXC_SEND_MSU();
                dv_LSC_state = STATE_LSC_IN_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_READY->IN_SERVICE");
             }
             else if (dv_LSC_state == STATE_LSC_ALIGNED_NOT_READY) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_IN_SERVICE,
own_id, iciptr_external);
                mtp2_timer_stop(&dv_timer1);
                dv_LSC_state = STATE_LSC_PROCESSOR_OUTAGE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_NOT_READY->PROCESSOR_OUTAGE");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                LSC_POC_REMOTE_PRO_RECOVERED();
```

```
                mtp2_send_external_msg(mtp3_id,
MSG_LSC_LSAC_REMOTE_PRO_RECOVERED, own_id, iciptr_external);
             }
             break;

          case MSG_RC_LSC_LINK_FAILURE:
             if (dv_LSC_state == STATE_LSC_INITIAL_ALIGNMENT) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_IAC_STOP();
                LSC_RC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
INITIAL_ALIGNMENT->OUT_SERVICE");
             }
             else if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
                mtp2_timer_stop(&dv_timer1);
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_RC_STOP();
                LSC_SUERM_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_READY->OUT_SERVICE, link failure declared by RC");
             }
             else if (dv_LSC_state == STATE_LSC_ALIGNED_NOT_READY) {
                mtp2_timer_stop(&dv_timer1);
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_RC_STOP();
                LSC_SUERM_STOP();
                LSC_TXC_SEND_SIOS();
                LSC_POC_STOP();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_NOT_READY->OUT_SERVICE, link failure declared by RC");
             }
             else if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->OUT_SERVICE, link failure declared by RC");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_POC_STOP();
                LSC_TXC_SEND_SIOS();
```

```
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
PROCESSOR_OUTAGE->OUT_SERVICE, link failure declared by RC");
             }
             break;

          case MSG_SUERM_LSC_LINK_FAILURE:
             if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
                mtp2_timer_stop(&dv_timer1);
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_RC_STOP();
                LSC_SUERM_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_READY->OUT_SERVICE, link failure declared by SUERM");
             }
             else if (dv_LSC_state == STATE_LSC_ALIGNED_NOT_READY) {
                mtp2_timer_stop(&dv_timer1);
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_RC_STOP();
                LSC_SUERM_STOP();
                LSC_TXC_SEND_SIOS();
                LSC_POC_STOP();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_NOT_READY->OUT_SERVICE, link failure declared by SUERM");
             }
             else if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->OUT_SERVICE, link failure declared by SUERM");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE,
own_id, iciptr_external);
                LSC_SUERM_STOP();
                LSC_RC_STOP();
                LSC_POC_STOP();
                LSC_TXC_SEND_SIOS();
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
PROCESSOR_OUTAGE->OUT_SERVICE, link failure declared by SUERM");
             }
             break;

          case MSG_POC_LSC_NO_PRO_OUTAGE:
```

```
            if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
               dv_LSC_processor_outage_during_service_flag = FALSE;
               if (dv_LSC_level3_indication_received_flag) {
                  dv_LSC_level3_indication_received_flag = FALSE;
                  LSC_TXC_SEND_MSU();
                  dv_LSC_local_processor_outage_during_alignment_flag =
FALSE;
                  LSC_RC_ACCEPT_MSU_FISU();
                  dv_LSC_state = STATE_LSC_IN_SERVICE;
                  op_prg_log_entry_write(log_hdle_link_status, "LSC: state
PROCESSOR_OUTAGE->IN_SERVICE");
               }
            }
            break;

         case MSG_LSAC_LSC_START:
            if (dv_LSC_state == STATE_LSC_OUT_SERVICE) {
               LSC_RC_START();
               LSC_TXC_START();
               if (dv_LSC_emergency_flag) {
                  LSC_IAC_EMERGENCY();
               }
               LSC_IAC_START();
               dv_LSC_state = STATE_LSC_INITIAL_ALIGNMENT;
               op_prg_log_entry_write(log_hdle_link_status, "LSC: state
OUT_SERVICE->INITIAL_ALIGNMENT, start link");
            }
            break;

         case MSG_LSAC_LSC_STOP:
            if (dv_LSC_state == STATE_LSC_INITIAL_ALIGNMENT) {
               LSC_IAC_STOP();
               LSC_RC_STOP();
               LSC_TXC_SEND_SIOS();
               dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
               dv_LSC_emergency_flag = FALSE;
               dv_LSC_state = STATE_LSC_OUT_SERVICE;
               op_prg_log_entry_write(log_hdle_link_status, "LSC: state
INITIAL_ALIGNMENT->OUT_SERVICE, STOP by LSAC");
            }
            else if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
               mtp2_timer_stop(&dv_timer1);
               LSC_RC_STOP();
               LSC_SUERM_STOP();
               LSC_TXC_SEND_SIOS();
               dv_LSC_emergency_flag = FALSE;
               dv_LSC_state = STATE_LSC_OUT_SERVICE;
               op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_READY->OUT_SERVICE, STOP by LSAC");
            }
            else if (dv_LSC_state == STATE_LSC_ALIGNED_NOT_READY) {
               mtp2_timer_stop(&dv_timer1);
               LSC_RC_STOP();
               LSC_SUERM_STOP();
               LSC_TXC_SEND_SIOS();
               LSC_POC_STOP();
               dv_LSC_emergency_flag = FALSE;
               dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
               dv_LSC_state = STATE_LSC_OUT_SERVICE;
               op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_NOT_READY->OUT_SERVICE, STOP by LSAC");
            }
```

```
            else if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
               LSC_SUERM_STOP();
               LSC_RC_STOP();
               LSC_TXC_SEND_SIOS();
               dv_LSC_emergency_flag = FALSE;
               dv_LSC_state = STATE_LSC_OUT_SERVICE;
               op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->OUT_SERVICE, STOP by LSAC");
            }
            else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
               LSC_SUERM_STOP();
               LSC_RC_STOP();
               LSC_POC_STOP();
               LSC_TXC_SEND_SIOS();
               dv_LSC_emergency_flag = FALSE;
               dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
               dv_LSC_state = STATE_LSC_OUT_SERVICE;
               op_prg_log_entry_write(log_hdle_link_status, "LSC: state
PROCESSOR_OUTAGE->OUT_SERVICE, STOP by LSAC");
            }
            break;

         case MSG_LSAC_LSC_EMERGENCY:
            if (dv_LSC_state == STATE_LSC_OUT_SERVICE) {
               dv_LSC_emergency_flag = TRUE;
            }
            else if (dv_LSC_state == STATE_LSC_INITIAL_ALIGNMENT) {
               dv_LSC_emergency_flag = TRUE;
               LSC_IAC_EMERGENCY();
            }
            break;

         case MSG_LSAC_LSC_EMERGENCY_CEASES:
            if (dv_LSC_state == STATE_LSC_OUT_SERVICE) {
               dv_LSC_emergency_flag = FALSE;
            }
            break;

         case MSG_LSAC_LSC_CONTINUE:
            if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
               dv_LSC_level3_indication_received_flag = TRUE;
               if (dv_LSC_processor_outage_during_service_flag) {
               }
               else {
                  dv_LSC_level3_indication_received_flag = FALSE;
                  LSC_TXC_SEND_MSU();
                  dv_LSC_local_processor_outage_during_alignment_flag =
FALSE;
                  LSC_RC_ACCEPT_MSU_FISU();
                  dv_LSC_state = STATE_LSC_IN_SERVICE;
                  op_prg_log_entry_write(log_hdle_link_status, "LSC: state
PROCESSOR_OUTAGE->IN_SERVICE");
               }
            }
            break;

         case MSG_LSAC_LSC_FLUSH_BUFFERS:
            if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
               LSC_TXC_FLUSH_BUFFERS();
               dv_LSC_level3_indication_received_flag = TRUE;
               if (dv_LSC_processor_outage_during_service_flag) {
               }
```

```
                else {
                   dv_LSC_level3_indication_received_flag = FALSE;
                   LSC_TXC_SEND_MSU();
                   dv_LSC_local_processor_outage_during_alignment_flag =
FALSE;
                   LSC_RC_ACCEPT_MSU_FISU();
                   dv_LSC_state = STATE_LSC_IN_SERVICE;
                   op_prg_log_entry_write(log_hdle_link_status, "LSC: state
PROCESSOR_OUTAGE->IN_SERVICE");
                }
             }
             break;

          case MSG_TCOC_LSC_RETRIEVE_BSNT:
             if (dv_LSC_state == STATE_LSC_OUT_SERVICE) {
                LSC_RC_RETRIEVE_BSNT();
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                LSC_RC_RETRIEVE_BSNT();
             }
             break;

          case MSG_TCOC_LSC_RETRIEVAL_REQ_FSNC:
             if (dv_LSC_state == STATE_LSC_OUT_SERVICE) {
                LSC_TXC_RETRIEVAL_REQ_FSNC();
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                LSC_TXC_RETRIEVAL_REQ_FSNC();
             }
             break;

          case MSG_MGMT_LSC_POWER_ON:
             if (dv_LSC_state == STATE_LSC_POWER_OFF) {
                LSC_TXC_START();
                LSC_TXC_SEND_SIOS();
                LSC_AERM_SET_Ti_To_Tin();
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_emergency_flag = FALSE;
                dv_LSC_state = STATE_LSC_OUT_SERVICE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
POWER_OFF->OUT_SERVICE");
             }
             break;

          case MSG_MGMT_LSC_LEVEL3_FAILURE:
             if (dv_LSC_state == STATE_LSC_OUT_SERVICE ) {
                dv_LSC_local_processor_outage_during_alignment_flag = TRUE;
             }
             else if (dv_LSC_state == STATE_LSC_INITIAL_ALIGNMENT) {
                dv_LSC_local_processor_outage_during_alignment_flag = TRUE;
             }
             else if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
                LSC_POC_LOCAL_PRO_OUTAGE();
                LSC_TXC_SEND_SIPO();
                LSC_RC_REJECT_MSU_FISU();
                dv_LSC_state = STATE_LSC_ALIGNED_NOT_READY;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_READY->ALIGNED_NOT_READY");
             }
             else if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
                LSC_POC_LOCAL_PRO_OUTAGE();
                LSC_TXC_SEND_SIPO();
```

```
                LSC_RC_REJECT_MSU_FISU();
                dv_LSC_processor_outage_during_service_flag = TRUE;
                dv_LSC_state = STATE_LSC_PROCESSOR_OUTAGE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->PROCESSOR_OUTAGE");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                LSC_POC_LOCAL_PRO_OUTAGE();
                LSC_TXC_SEND_SIPO();
             }
             break;

          case MSG_MGMT_LSC_LOCAL_PRO_OUTAGE:
             if (dv_LSC_state == STATE_LSC_OUT_SERVICE) {
                dv_LSC_local_processor_outage_during_alignment_flag = TRUE;
             }
             else if (dv_LSC_state == STATE_LSC_INITIAL_ALIGNMENT) {
                dv_LSC_local_processor_outage_during_alignment_flag = TRUE;
             }
             else if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
                LSC_POC_LOCAL_PRO_OUTAGE();
                LSC_TXC_SEND_SIPO();
                LSC_RC_REJECT_MSU_FISU();
             }
             else if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
                LSC_POC_LOCAL_PRO_OUTAGE();
                LSC_TXC_SEND_SIPO();
                LSC_RC_REJECT_MSU_FISU();
                dv_LSC_processor_outage_during_service_flag = TRUE;
                dv_LSC_state = STATE_LSC_PROCESSOR_OUTAGE;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->PROCESSOR_OUTAGE");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                LSC_POC_LOCAL_PRO_OUTAGE();
                LSC_TXC_SEND_SIPO();
             }
             break;

          case MSG_MGMT_LSC_LOCAL_PRO_RECOVERED:
             if (dv_LSC_state == STATE_LSC_OUT_SERVICE) {
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
             }
             else if (dv_LSC_state == STATE_LSC_INITIAL_ALIGNMENT) {
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                dv_LSC_state = STATE_LSC_ALIGNED_READY;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
INITIAL_ALIGNMENT->ALIGNED_READY");
             }
             else if (dv_LSC_state == STATE_LSC_ALIGNED_NOT_READY) {
                LSC_POC_LOCAL_PRO_RECOVERED();
                dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
                LSC_TXC_SEND_FISU();
                LSC_RC_ACCEPT_MSU_FISU();
                dv_LSC_state = STATE_LSC_ALIGNED_READY;
                op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_NOT_READY->ALIGNED_READY");
             }
             else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
                LSC_POC_LOCAL_PRO_RECOVERED();
                LSC_RC_RETRIEVE_FSNX();
                LSC_TXC_SEND_FISU();
```

```
            }
            break;

         case MSG_LSAC_LSC_LOCAL_PRO_OUTAGE:
            if (dv_LSC_state == STATE_LSC_OUT_SERVICE) {
               dv_LSC_local_processor_outage_during_alignment_flag = TRUE;
            }
            else if (dv_LSC_state == STATE_LSC_INITIAL_ALIGNMENT) {
               dv_LSC_local_processor_outage_during_alignment_flag = TRUE;
            }
            else if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
               LSC_POC_LOCAL_PRO_OUTAGE();
               LSC_TXC_SEND_SIPO();
               LSC_RC_REJECT_MSU_FISU();
               dv_LSC_state = STATE_LSC_ALIGNED_NOT_READY;
               op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_READY->ALIGNED_NOT_READY");
            }
            else if (dv_LSC_state == STATE_LSC_IN_SERVICE) {
               LSC_POC_LOCAL_PRO_OUTAGE();
               LSC_TXC_SEND_SIPO();
               LSC_RC_REJECT_MSU_FISU();
               dv_LSC_processor_outage_during_service_flag = TRUE;
               dv_LSC_state = STATE_LSC_PROCESSOR_OUTAGE;
               op_prg_log_entry_write(log_hdle_link_status, "LSC: state
IN_SERVICE->PROCESSOR_OUTAGE");
            }
            else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
               LSC_POC_LOCAL_PRO_OUTAGE();
               LSC_TXC_SEND_SIPO();
            }
            break;

         case MSG_LSAC_LSC_LOCAL_PRO_RECOVERED:
            if (dv_LSC_state == STATE_LSC_OUT_SERVICE) {
               dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
            }
            else if (dv_LSC_state == STATE_LSC_INITIAL_ALIGNMENT) {
               dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
               dv_LSC_state = STATE_LSC_ALIGNED_READY;
               op_prg_log_entry_write(log_hdle_link_status, "LSC: state
INITAL_ALIGNMENT->ALIGNED_READY");
            }
            else if (dv_LSC_state == STATE_LSC_ALIGNED_NOT_READY) {
               LSC_POC_LOCAL_PRO_RECOVERED();
               dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
               LSC_TXC_SEND_FISU();
               LSC_RC_ACCEPT_MSU_FISU();
               dv_LSC_state = STATE_LSC_ALIGNED_READY;
               op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_NOT_READY->ALIGNED_READY");
            }
            else if (dv_LSC_state == STATE_LSC_PROCESSOR_OUTAGE) {
               LSC_POC_LOCAL_PRO_RECOVERED();
               LSC_RC_RETRIEVE_FSNX();
               LSC_TXC_SEND_FISU();
            }
            break;
      }
      op_pk_destroy(pkptr);
   }
```

```
      /* schedule next internal message processing interrupt */
      op_intrpt_schedule_self(op_sim_time() + LSC_service_time, LSC_MSG);
      break;

   case T1_EXPIRE:

      /* process expiry of T1, Timer "Alignment Ready" */
      /* Timer which limits the time LSC has to change */
      /* to Service once alignment is ready, else LSC  */
      /* will go back to Out of Service                */
      if (dv_LSC_state == STATE_LSC_ALIGNED_READY) {
         mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE, 0,
iciptr_external);
         LSC_RC_STOP();
         LSC_SUERM_STOP();
         LSC_TXC_SEND_SIOS();
         dv_LSC_emergency_flag = FALSE;
         dv_LSC_state = STATE_LSC_OUT_SERVICE;
         op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_READY->OUT_SERVICE");
      }
      else if (dv_LSC_state == STATE_LSC_ALIGNED_NOT_READY) {
         mtp2_send_external_msg(mtp3_id, MSG_LSC_LSAC_OUT_SERVICE, 0,
iciptr_external);
         LSC_RC_STOP();
         LSC_SUERM_STOP();
         LSC_TXC_SEND_SIOS();
         LSC_POC_STOP();
         dv_LSC_emergency_flag = FALSE;
         dv_LSC_local_processor_outage_during_alignment_flag = FALSE;
         dv_LSC_state = STATE_LSC_OUT_SERVICE;
         op_prg_log_entry_write(log_hdle_link_status, "LSC: state
ALIGNED_NOT_READY->OUT_SERVICE");
      }
      break;
}
```

M. EXECUTIVE FOR "IAC" STATE

```
/* ================================================================== */
/* Initial Alignment Control (IAC)                                    */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* Note:                                                              */
/*    This defines all IAC processing which includes handling of      */
/*    incoming internal messages and timer expiry interrupts.         */
/*    Messages are proccessed in FIFO order.                          */
/*    Each message is processed according to the algorithm            */
/*    as defined in ITU-T Q.703 (Jul 96), Fig 9.                      */
/* ================================================================== */


switch (op_intrpt_code()) {
   case IAC_MSG:

      /* If incoming message exists, remove the message packet */
      /* from the buffer and process it                        */
      /* The first field in each packet is always the message  */
      /* code, it can be followed by one or more fields        */
      /* depending on the message                              */

      if (!op_subq_empty(IAC_BUFFER)) {
         pkptr = op_subq_pk_remove(IAC_BUFFER, OPC_QPOS_HEAD);
         op_pk_fd_get(pkptr, FD_INDEX_1ST, &msg_code);
         switch (msg_code) {

            case MSG_LSC_IAC_START:

               /* start proving by sending out SIO and activate timer2 */
               if (dv_IAC_state == STATE_IAC_IDLE) {
                  IAC_TXC_SEND_SIO();
                  mtp2_timer_start(&dv_timer2);
                  dv_IAC_state = STATE_IAC_NOT_ALIGNED;
                  op_prg_log_entry_write(log_hdle_link_status, "IAC: state
IDLE->NOT_ALIGNED");
               }
               break;

            case MSG_LSC_IAC_STOP:
               if (dv_IAC_state == STATE_IAC_NOT_ALIGNED) {
                  mtp2_timer_stop(&dv_timer2);
                  dv_IAC_emergency_flag = FALSE;
                  dv_IAC_state = STATE_IAC_IDLE;
                  op_prg_log_entry_write(log_hdle_link_status, "IAC: state
NOT_ALIGNED->IDLE");
               }
               else if (dv_IAC_state == STATE_IAC_ALIGNED) {
                  mtp2_timer_stop(&dv_timer3);
                  dv_IAC_emergency_flag = FALSE;
                  dv_IAC_state = STATE_IAC_IDLE;
                  op_prg_log_entry_write(log_hdle_link_status, "IAC: state
ALIGNED->IDLE");
               }
               else if (dv_IAC_state == STATE_IAC_PROVING) {
                  mtp2_timer_stop(&dv_timer4);
                  IAC_AERM_STOP();
                  dv_IAC_emergency_flag = FALSE;
                  dv_IAC_state = STATE_IAC_IDLE;
```

```
                op_prg_log_entry_write(log_hdle_link_status, "IAC: state
PROVING->IDLE");
                }
                break;

            case MSG_LSC_IAC_EMERGENCY:
                if (dv_IAC_state == STATE_IAC_IDLE) {
                   dv_IAC_emergency_flag = TRUE;
                }
                else if (dv_IAC_state == STATE_IAC_NOT_ALIGNED) {
                   dv_IAC_emergency_flag = TRUE;
                }
                else if (dv_IAC_state == STATE_IAC_ALIGNED) {
                   IAC_TXC_SEND_SIE();
                   dv_timer4.delay = T4_Pe;
                }
                else if (dv_IAC_state == STATE_IAC_PROVING) {
                   IAC_TXC_SEND_SIE();
                   mtp2_timer_stop(&dv_timer4);
                   dv_timer4.delay = T4_Pe;
                   IAC_AERM_STOP();
                   IAC_AERM_SET_Ti_To_Tie();
                   IAC_AERM_START();
                   dv_IAC_further_proving_flag = FALSE;
                   mtp2_timer_start(&dv_timer4);
                }
                break;

            case MSG_RC_IAC_SIO:

                /* if IAC is not aligned yet, assume alignment and */
                /* start T3 else go back to not aligned if already */
                /* aligned                                         */
                if (dv_IAC_state == STATE_IAC_NOT_ALIGNED) {
                   mtp2_timer_stop(&dv_timer2);
                   if (dv_IAC_emergency_flag) {
                      dv_timer4.delay = T4_Pe;
                      IAC_TXC_SEND_SIE();
                   }
                   else {
                      dv_timer4.delay = T4_Pn;
                      IAC_TXC_SEND_SIN();
                   }
                   mtp2_timer_start(&dv_timer3);
                   dv_IAC_state = STATE_IAC_ALIGNED;
                   op_prg_log_entry_write(log_hdle_link_status, "IAC: state
NOT_ALIGNED->ALIGNED");
                }
                else if (dv_IAC_state == STATE_IAC_PROVING) {
                   mtp2_timer_stop(&dv_timer4);
                   IAC_AERM_STOP();
                   mtp2_timer_start(&dv_timer3);
                   dv_IAC_state = STATE_IAC_ALIGNED;
                   op_prg_log_entry_write(log_hdle_link_status, "IAC: state
PROVING->ALIGNED");
                }
                break;

            case MSG_RC_IAC_SIN:

                /* if not aligned yet assume aligned and start timer3  */
                /* if already aligned, assume proving and start timer4 */
```

```
            if (dv_IAC_state == STATE_IAC_NOT_ALIGNED) {
               mtp2_timer_stop(&dv_timer2);
               if (dv_IAC_emergency_flag) {
                  dv_timer4.delay = T4_Pe;
                  IAC_TXC_SEND_SIE();
               }
               else {
                  dv_timer4.delay = T4_Pn;
                  IAC_TXC_SEND_SIN();
               }
               mtp2_timer_start(&dv_timer3);
               dv_IAC_state = STATE_IAC_ALIGNED;
               op_prg_log_entry_write(log_hdle_link_status, "IAC: state
NOT_ALIGNED->ALIGNED");
            }
            else if (dv_IAC_state == STATE_IAC_ALIGNED) {
               mtp2_timer_stop(&dv_timer3);
               if (dv_timer4.delay == T4_Pe) {
                  IAC_AERM_SET_Ti_To_Tie();
               }
               IAC_AERM_START();
               mtp2_timer_start(&dv_timer4);
               dv_IAC_Cp = 0;
               dv_IAC_further_proving_flag = FALSE;
               dv_IAC_state = STATE_IAC_PROVING;
               op_prg_log_entry_write(log_hdle_link_status, "IAC: state
ALIGNED->PROVING");
            }
            break;

         case MSG_RC_IAC_SIE:
            if (dv_IAC_state == STATE_IAC_NOT_ALIGNED) {
               mtp2_timer_stop(&dv_timer2);
               dv_timer4.delay = T4_Pe;
               if (dv_IAC_emergency_flag) {
                  IAC_TXC_SEND_SIE();
               }
               else {
                  IAC_TXC_SEND_SIN();
               }
               mtp2_timer_start(&dv_timer3);
               dv_IAC_state = STATE_IAC_ALIGNED;
               op_prg_log_entry_write(log_hdle_link_status, "IAC: state
NOT_ALIGNED->ALIGNED");
            }
            else if (dv_IAC_state == STATE_IAC_ALIGNED) {
               dv_timer4.delay = T4_Pe;
               mtp2_timer_stop(&dv_timer3);
               IAC_AERM_SET_Ti_To_Tie();
               IAC_AERM_START();
               mtp2_timer_start(&dv_timer4);
               dv_IAC_Cp = 0;
               dv_IAC_further_proving_flag = FALSE;
               dv_IAC_state = STATE_IAC_PROVING;
               op_prg_log_entry_write(log_hdle_link_status, "IAC: state
ALIGNED->PROVING");
            }
            else if (dv_IAC_state == STATE_IAC_PROVING) {
               if (dv_timer4.delay != T4_Pe) {
                  mtp2_timer_stop(&dv_timer4);
                  dv_timer4.delay = T4_Pe;
                  IAC_AERM_STOP();
```

```
                    IAC_AERM_SET_Ti_To_Tie();
                    IAC_AERM_START();
                    dv_IAC_further_proving_flag = FALSE;
                    mtp2_timer_start(&dv_timer4);
                 }
              }
              break;

           case MSG_RC_IAC_SIOS:

              /* opposite end has gone out of service */
              /* alignment is not possible           */
              if (dv_IAC_state == STATE_IAC_ALIGNED) {
                 IAC_LSC_ALIGNMENT_NOT_POSSIBLE();
                 mtp2_timer_stop(&dv_timer3);
                 dv_IAC_emergency_flag = FALSE;
                 dv_IAC_state = STATE_IAC_IDLE;
                 op_prg_log_entry_write(log_hdle_link_status, "IAC: state
ALIGNED->IDLE");
              }
              else if (dv_IAC_state == STATE_IAC_PROVING) {
                 mtp2_timer_stop(&dv_timer4);
                 IAC_LSC_ALIGNMENT_NOT_POSSIBLE();
                 IAC_AERM_STOP();
                 dv_IAC_emergency_flag = FALSE;
                 dv_IAC_state = STATE_IAC_IDLE;
                 op_prg_log_entry_write(log_hdle_link_status, "IAC: state
PROVING->IDLE");
              }
              break;

           case MSG_AERM_IAC_ABORT_PROVING:

              /* too much link error detected, cancel current proving */
              /* start another proving period unless this is the 5th  */
              /* consecutive attempt already. Then alignment is not   */
              /* possible                                             */
              if (dv_IAC_state == STATE_IAC_PROVING) {
                 dv_IAC_Cp = dv_IAC_Cp + 1;
                 if (dv_IAC_Cp == 5) {
                    IAC_LSC_ALIGNMENT_NOT_POSSIBLE();
                    mtp2_timer_stop(&dv_timer4);
                    IAC_AERM_STOP();
                    dv_IAC_emergency_flag = FALSE;
                    dv_IAC_state = STATE_IAC_IDLE;
                    op_prg_log_entry_write(log_hdle_link_status, "IAC: state
PROVING->IDLE");
                 }
                 else {
                    dv_IAC_further_proving_flag = TRUE;
                 }
              }
              break;

           case MSG_DAEDR_IAC_CORRECT_SU:

              /* restart proving when AERM has recovered from failure */
              if (dv_IAC_state == STATE_IAC_PROVING) {
                 if (dv_IAC_further_proving_flag) {
                    mtp2_timer_stop(&dv_timer4);
                    IAC_AERM_START();
                    dv_IAC_further_proving_flag = FALSE;
```

```
                    mtp2_timer_start(&dv_timer4);
                }
            }
            break;
        }
        op_pk_destroy(pkptr);
    }
    /* schedule next internal message processing interrupt */
    op_intrpt_schedule_self(op_sim_time() + IAC_service_time, IAC_MSG);
    break;

  case T2_EXPIRE:

    /* process expiry of T2, timer "not aligned" */
    /* alignment not possible as LSC cannot get aligned */
    /* before timer expire                              */
    if (dv_IAC_state == STATE_IAC_NOT_ALIGNED) {
        IAC_LSC_ALIGNMENT_NOT_POSSIBLE();
        dv_IAC_emergency_flag = FALSE;
        dv_IAC_state = STATE_IAC_IDLE;
        op_prg_log_entry_write(log_hdle_link_status, "IAC: state NOT_ALIGNED-
>IDLE, T2 Expired");
    }
    break;

  case T3_EXPIRE:

    /* Process expiry of T3, timer "aligned"            */
    /* Both ends cannot achieved alignment before timer */
    /* expires alignment not possible                   */
    if (dv_IAC_state == STATE_IAC_ALIGNED) {
        IAC_LSC_ALIGNMENT_NOT_POSSIBLE();
        dv_IAC_emergency_flag = FALSE;
        dv_IAC_state = STATE_IAC_IDLE;
        op_prg_log_entry_write(log_hdle_link_status, "IAC: state ALIGNED->IDLE,
T3 Expired");
    }
    break;

  case T4_EXPIRE:

    /* process expiry of T4, Proving period timer */
    /* if not further proving during the period   */
    /* alignment is complete else go for another  */
    /* proving period by AERM                     */
    if (dv_IAC_state == STATE_IAC_PROVING) {
        if (dv_IAC_further_proving_flag) {
            IAC_AERM_START();
            dv_IAC_further_proving_flag = FALSE;
            mtp2_timer_start(&dv_timer4);
        }
        else {
            IAC_LSC_ALIGNMENT_COMPLETE();
            IAC_AERM_STOP();
            dv_IAC_emergency_flag = FALSE;
            dv_IAC_state = STATE_IAC_IDLE;
            op_prg_log_entry_write(log_hdle_link_status, "IAC: state PROVING-
>IDLE, alignment complete");
        }
    }
    break;
}
```

N. EXECUTIVE FOR "POC" STATE

```
/* ================================================================== */
/* Processor Outage Control (POC)                                     */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* Note:                                                              */
/*    This defines all POC processing which includes handling of      */
/*    incoming internal messages.                                     */
/*    Messages are proccessed in FIFO order.                          */
/*    Each message is processed according to the algorithm            */
/*    as defined in ITU-T Q.703 (Jul 96), Fig 10.                     */
/* ================================================================== */


/* If incoming message exists, remove the message packet */
/* from the buffer and process it                        */
/* The first field in each packet is always the message  */
/* code, it can be followed by one or more fields        */
/* depending on the message                              */

if (!op_subq_empty(POC_BUFFER)) {
   pkptr = op_subq_pk_remove(POC_BUFFER, OPC_QPOS_HEAD);
   op_pk_fd_get(pkptr, FD_INDEX_1ST, &msg_code);
   switch (msg_code) {

      case MSG_LSC_POC_LOCAL_PRO_OUTAGE:
         if (dv_POC_state == STATE_POC_IDLE ) {
            dv_POC_state = STATE_POC_LOCAL_PRO_OUTAGE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state IDLE->LOCAL_PRO_OUTAGE, local pro outage as inform
by LSC");
         }
         else if (dv_POC_state == STATE_POC_REMOTE_PRO_OUTAGE) {
            dv_POC_state =STATE_POC_BOTH_PRO_OUT;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state REMOTE_PRO_OUTAGE->BOTH_PRO_OUT, local pro outage as
inform by LSC");
         }
         break;

      case MSG_LSC_POC_LOCAL_PRO_RECOVERED:
         if (dv_POC_state == STATE_POC_LOCAL_PRO_OUTAGE ) {
            POC_LSC_NO_PRO_OUTAGE();
            dv_POC_state = STATE_POC_IDLE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state LOCAL_PRO_OUT->IDLE, local pro recovered as inform
by LSC");
         }
         else if (dv_POC_state == STATE_POC_BOTH_PRO_OUT) {
            dv_POC_state = STATE_POC_REMOTE_PRO_OUTAGE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state BOTH_PRO_OUT->REMOTE_PRO_OUTAGE, local pro recovered
as inform by LSC");
         }
         break;

          case MSG_LSC_POC_REMOTE_PRO_OUTAGE:
```

```
         if (dv_POC_state == STATE_POC_IDLE ) {
            dv_POC_state = STATE_POC_REMOTE_PRO_OUTAGE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state IDLE->REMOTE_PRO_OUT, remote pro outage as inform by
LSC");
                }
         else if (dv_POC_state == STATE_POC_LOCAL_PRO_OUTAGE ) {
                        dv_POC_state = STATE_POC_BOTH_PRO_OUT;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state LOCAL_PRO_OUT->BOTH_PRO_OUT, remote pro outage as
inform by LSC");
         }
         break;

      case MSG_LSC_POC_REMOTE_PRO_RECOVERED:
         if (dv_POC_state == STATE_POC_BOTH_PRO_OUT) {
            dv_POC_state = STATE_POC_LOCAL_PRO_OUTAGE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state BOTH_PRO_OUT->LOCAL_PRO_OUT, remote pro recovered as
inform by LSC");
         }
         else if (dv_POC_state == STATE_POC_REMOTE_PRO_OUTAGE) {
            POC_LSC_NO_PRO_OUTAGE();
            dv_POC_state = STATE_POC_IDLE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state REMOTE_PRO_OUT->IDLE, remote pro recovered as inform
by LSC");
         }
         break;

      case MSG_LSC_POC_STOP:
         if (dv_POC_state == STATE_POC_LOCAL_PRO_OUTAGE ) {
            dv_POC_state = STATE_POC_IDLE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state LOCAL_PRO_OUT->IDLE, STOP by LSC");
         }
         else if (dv_POC_state == STATE_POC_BOTH_PRO_OUT) {
            dv_POC_state = STATE_POC_IDLE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state BOTH_PRO_OUT->IDLE, STOP by LSC");
         }
         else if (dv_POC_state == STATE_POC_REMOTE_PRO_OUTAGE) {
            dv_POC_state = STATE_POC_IDLE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "POC: state REMOTE_PRO_OUT->IDLE, STOP by LSC");
         }
         break;
   }
   op_pk_destroy(pkptr);
}

/* schedule next internal message processing interrupt */
op_intrpt_schedule_self(op_sim_time() + POC_service_time, POC_MSG);
```

O. EXECUTIVE FOR "AERM" STATE

```
/* ================================================================== */
/* Alignment Error Rate Monitor (AERM)                                */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* Note:                                                              */
/*    This defines all AERM processing which includes handling of     */
/*    incoming internal messages                                      */
/*    Messages are proccessed in FIFO order.                          */
/*    Each message is processed according to the algorithm            */
/*    as defined in ITU-T Q.703 (Jul 96), Fig 17.                     */
/* ================================================================== */


/* If incoming message exists, remove the message packet */
/* from the buffer and process it                        */
/* The first field in each packet is always the message  */
/* code, it can be followed by one or more fields        */
/* depending on the message                              */

if (!op_subq_empty(AERM_BUFFER)) {
   pkptr = op_subq_pk_remove(AERM_BUFFER, OPC_QPOS_HEAD);
   op_pk_fd_get(pkptr, FD_INDEX_1ST, &msg_code);
   switch (msg_code) {

      case MSG_LSC_AERM_SET_Ti_To_Tin:
         if (dv_AERM_state == STATE_AERM_IDLE) {
            dv_AERM_Ti = Tin;
         }
         break;

      case MSG_IAC_AERM_SET_Ti_To_Tie:
         if (dv_AERM_state == STATE_AERM_IDLE) {
            dv_AERM_Ti = Tie;
         }
         break;

          case MSG_IAC_AERM_START:
         if (dv_AERM_state == STATE_AERM_IDLE) {
            dv_AERM_Ca = 0;
                       dv_AERM_state = STATE_AERM_MONITORING;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "AERM: state IDLE->MONITORING, start error monitoring as
instructed by IAC");
         }
         break;

      case MSG_IAC_AERM_STOP:
         if (dv_AERM_state == STATE_AERM_MONITORING) {
            dv_AERM_Ti = Tin;
            dv_AERM_state = STATE_AERM_IDLE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "AERM: state MONITORING->IDLE, STOP as instructed by IAC");
         }
         break;

      case MSG_DAEDR_AERM_SU_ERROR:

         /* Increase one count everytime a error SU is recieved          */
```

```
         /* DAEDR sends an error SU message when a Error SU is received   */
         /* and also for every 16 octets received when in "octet counting */
         /* mode.                                                         */
         if (dv_AERM_state == STATE_AERM_MONITORING) {
            dv_AERM_Ca = dv_AERM_Ca + 1;
                        if (dv_AERM_Ca == dv_AERM_Ti) {
               AERM_IAC_ABORT_PROVING();
               dv_AERM_state = STATE_AERM_IDLE;
               op_prg_log_entry_write(
                  log_hdle_link_status,
                  "AERM: state MONITORING->IDLE, proving fail as too many
errors detected by DAEDR");
             }
         }
         break;
   }
   op_pk_destroy(pkptr);
}

/* schedule next internal message processing interrupt */
op_intrpt_schedule_self(op_sim_time() + AERM_service_time, AERM_MSG);
```

P. EXECUTIVE FOR "SUERM" STATE

```
/* ========================================================================== */
/* Signal Unit Error Rate Monitor (SUERM)                                     */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)             */
/* Note:                                                                      */
/*    This defines all SUERM processing which includes handling of            */
/*    incoming internal messages                                              */
/*    Messages are proccessed in FIFO order.                                  */
/*    Each message is processed according to the algorithm                    */
/*    as defined in ITU-T Q.703 (Jul 96), Fig 18.                             */
/* ========================================================================== */


/* If incoming message exists, remove the message packet */
/* from the buffer and process it                        */
/* The first field in each packet is always the message  */
/* code, it can be followed by one or more fields        */
/* depending on the message                              */

if (!op_subq_empty(SUERM_BUFFER)) {
   pkptr = op_subq_pk_remove(SUERM_BUFFER, OPC_QPOS_HEAD);
   op_pk_fd_get(pkptr, FD_INDEX_1ST, &msg_code);
   switch (msg_code) {

      case MSG_LSC_SUERM_START:
         if (dv_SUERM_state == STATE_AERM_IDLE) {
            dv_SUERM_Cs = 0;
                   dv_SUERM_Nsu = 0;
            dv_SUERM_state = STATE_SUERM_IN_SERVICE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "SUERM: state IDLE->IN_SERVICE, START as instructed by LSC");
         }
         break;

      case MSG_LSC_SUERM_STOP:
         if (dv_SUERM_state == STATE_AERM_IN_SERVICE) {
            dv_SUERM_state = STATE_SUERM_IDLE;
            op_prg_log_entry_write(
               log_hdle_link_status,
               "SUERM: state IN_SERVICE->IDLE, STOP as instructed by LSC");
         }
         break;

      case MSG_DAEDR_SUERM_SU_ERROR:

         /* declare error when count reaches T                        */
         /* DAEDR sends SU error messages when an SU is in error      */
         /* or whenever 16 octets is received during "octet counting */
         /* mode". Count is decremented by one whenever 256 SUs are   */
         /* received whenter error or not                             */
         if (dv_SUERM_state == STATE_AERM_IN_SERVICE) {
            dv_SUERM_Cs = dv_SUERM_Cs + 1;
                   dv_SUERM_Nsu = dv_SUERM_Nsu + 1;
            if (dv_SUERM_Cs == T) {
                         SUERM_LSC_LINK_FAILURE();
               dv_SUERM_state = STATE_SUERM_IDLE;
               op_prg_log_entry_write(
                  log_hdle_link_status,
```

```
                    "SUERM: state IN_SERVICE->IDLE, too many errors as detected
by DAEDR");
                 }
                     else {
                if (dv_SUERM_Nsu == 256) {
                                dv_SUERM_Nsu = 0;
                   if (dv_SUERM_Cs != 0) {
                                   dv_SUERM_Cs = dv_SUERM_Cs - 1;
                                 }
                }
              }
          }
          break;

       case MSG_DAEDR_SUERM_CORRECT_SU:
          if (dv_SUERM_state == STATE_AERM_IN_SERVICE) {
             dv_SUERM_Nsu = dv_SUERM_Nsu + 1;
                    if (dv_SUERM_Nsu == 256) {
                       dv_SUERM_Nsu = 0;
                if (dv_SUERM_Cs != 0) {
                           dv_SUERM_Cs = dv_SUERM_Cs - 1;
                            }
             }
          }
          break;
    }
    op_pk_destroy(pkptr);
}

/* schedule next internal message processing interrupt */
op_intrpt_schedule_self(op_sim_time() + SUERM_service_time, SUERM_MSG);
```

Q. EXECUTIVE FOR "CC" STATE

```
/* ================================================================== */
/* Congestion Control (CC)                                            */
/* Signaling System 7 (SS7), Message Transfer Part Level 2 (MTP2)     */
/* Note:                                                              */
/*    This defines all CC processing which includes handling of       */
/*    incoming internal messages and timer expiry.                    */
/*    Messages are proccessed in FIFO order.                          */
/*    Each message is processed according to the algorithm            */
/*    as defined in ITU-T Q.703 (Jul 96), Fig 19.                     */
/* ================================================================== */

switch (op_intrpt_code()) {
   case CC_MSG:
      /* If incoming message exists, remove the message packet */
      /* from the buffer and process it                        */
      /* The first field in each packet is always the message  */
      /* code, it can be followed by one or more fields        */
      /* depending on the message                              */

      if (!op_subq_empty(CC_BUFFER)) {
         pkptr = op_subq_pk_remove(CC_BUFFER, OPC_QPOS_HEAD);
         op_pk_fd_get(pkptr, FD_INDEX_1ST, &msg_code);
         switch (msg_code) {

          case MSG_RC_CC_NORMAL:
               if (dv_CC_state == STATE_CC_LEVEL2_CONGESTION) {
                  mtp2_timer_stop(&dv_timer5);
                  dv_CC_state = STATE_CC_IDLE;
                  op_prg_log_entry_write(
                     log_hdle_link_status,
                     "CC: state LEVEL2_CONGESTION->IDLE, as inform by RC");
               }
               break;

            case MSG_RC_CC_BUSY:
               if (dv_CC_state == STATE_CC_IDLE ) {
                  CC_TXC_SEND_SIB();
                  mtp2_timer_start(&dv_timer5);
                  dv_CC_state = STATE_CC_LEVEL2_CONGESTION;
                  op_prg_log_entry_write(
                     log_hdle_link_status,
                     "CC: state IDLE->LEVEL2_CONGESTION, as inform by RC");
               }
               break;
         }
         op_pk_destroy(pkptr);
      }
      /* schedule next internal message processing interrupt */
      op_intrpt_schedule_self(op_sim_time() + CC_service_time, CC_MSG);
      break;
   case T5_EXPIRE:
      /* process T5 expiry, Timer "sending SIB" */
      if (dv_CC_state == STATE_CC_LEVEL2_CONGESTION) {
         CC_TXC_SEND_SIB();
         mtp2_timer_start(&dv_timer5);
      }
      break;
}
```

THIS PAGE IS INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Willmann, G. and Kuhn, P.J., "Performance Modeling of Signaling System No.7," *IEEE Communications Magazine*, Vol. 28, pp. 44-56, Jul 1991

[2] Unger, B.W., Goetz, D.J., Maryka, S.W., " Simulation of SS7 Common Channel Signaling," *IEEE Communications Magazine*, Vol. 32, pp. 52-62, Mar 1994

[3] International Telecommunications Union Telecommunications Recommendation, ITU-T Q.701, *Introduction to CCITT Signaling System No. 7*, Mar 1993

[4] International Telecommunications Union Telecommunications Recommendation, ITU-T Q.703, *Functional Description of Message Transfer Part (MTP) of Signaling System No. 7*, Jul 1996

[5] International Telecommunications Union Telecommunications Recommendation, ITU-T Q.704, *Functional Description of Message Transfer Part (MTP) of Signaling System No. 7*, Jul 1996

[6] Ow, K.C., *OPNET Simulation of Signaling System No. 7 Network Interfaces*, Master's Thesis, Naval Postgraduate School, Monterey, California, Mar 2000

[7] Wong, J. L., "Traffic Routing and Performance Analysis of the Common Channel Signaling System 7 Network," *IEEE Communications Magazine*, Jan 1991

[8] Schwartz, M., *Broadband Integrated Networks*, pp.342, 353, Prentice Hall, New York, 1996

THIS PAGE IS INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   8725 John J. Kingman Rd., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library..... 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code EC..... 1
   Department of Electical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

4. Prof. John McEachen, Code EC/Mj.....1
   Department of Electical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

5. Prof. Murali Tummala, Code EC/Tu.....1
   Department of Electical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

6. Naval Engineering Logistics Office.....1
   P.O. Box 15467
   Arlington, VA 22215-5000

7. Ms. Rosemary Wenchel..... 1
   Naval Information Warfare Analysis Center
   9800 Savage Road
   Ft. Meade, MD 20755

8. Dr. William Semancik.....1
   Laboratory for Telecommunications Science
   9800 Savage Road
   Ft. Meade, MD 20755
   ATTN: R56

9. Head Librarian.....1
   Defence Technology Tower Library
   1 Depot Rd., #02-01
   Defence Technology Tower A
   Singapore 109681

10. Lim Chin Thong.....1
Blk. 15, #07-103,
Hougang Ave 3,
Singapore 530015

69 290NPG TH 2971
6/02 22527-200 NLB

DUDLEY KNOX LIBRARY
3 2768 00403137 7